ON THE CONSTRUCTION OF A DEEP LEARNING ARCHITECTURE

Jimmy Tsz Ming Yue
School of Computer Science
University of Sydney
jyue6728@uni.sydney.edu.au
SID : 440159151

Melissa Tan
School of Computer Science
University of Sydney
mtan2988@uni.sydney.edu.au
SID : 200249191

Zhuoyang Li
School of Computer Science
University of Sydney
zhli2344@uni.sydney.edu.au
SID : 480164337

May 3, 2019

1 Introduction

1.1 Motivation and Overview

The ability of utilising Back-propagating Neural Networks in generating accurate classifications of multiple data-sets has been widely discussed within literature [1], [2], [3]. The performance of such Networks are varied within individual implementations utilising differing regularisation and optimisation methods. Such methods not only improve the accuracy of classification, but also heightens the algorithmic efficiency through reducing the computational time required for generating multi-class labels. In this study, we design and implement a Neural Network utilising deep learning structures such as mini-batching, stochastic gradient descent, weight decay and dropout to classify an unknown data-set that is provided. Through such an implementation, it is possible to investigate the importance and influence of the network's structures such as varying optimisation and regularisation methods on classification performance. By understanding the theoretical aspects of these methods, which we present in our Methodology and evaluate the effects of each component on its success of generating class labels through validation with a test set, we are able to consider which Deep learning methods to further employ in future studies.

1.2 Dataset

The given dataset consists of $60,000$ examples provided for training along with a further $10,000$ test data with 10 distinct labels from 0 to 9. The label distribution of the training set given is equally distributed, at $1,000$ samples per class as such, there is no indication of any class imbalance which may generate model misrepresentation. Therefore, we conclude that there is unnecessary to conduct any re-sampling approaches such as over-sampling, under-sampling and SMOTE [4] sampling to address class imbalance issues.

2 Methods

2.1 Pre-processing

Research from [5] shows that feature normalisation significantly improves classification's accuracy. As each feature may have varying units, magnitude and range, the results would vary significantly between the features. To counter this, we employed the Z-score standardisation method

$$x = \frac{x - \bar{x}}{\sigma} \tag{1}$$

to normalise samples within the dataset in order to yield higher levels of performance.

2.2 Multi-Layer Networks

A multi-layer artificial Neural Network (ANN) is an emulation of the organic composition from a biological brain cellular network. In such a consideration, algorithms that are constructed possess individual "Neurons" [6] are arranged in a manner such that they are joined together in the form of communication pathways that seek to carry out computations heuristically. Networks that are created for this purpose are represented mathematically as graphs with neurons represented by nodal objects and associated directed edges between other neurons. Such directed edges indicate the flow of information within the network. To formally denote such a structure, we consider the neural network as the directed acylic graph, $G = (V, E)$, where the sets of vertices V and edges E are neurons and pathways respectively. Then let $w : E \to \mathbb{C}$ be a weight function over the edges, and $\sigma : \mathbb{R} \to \mathbb{R}$ be functions that model the operation of a singular neuron. In traditionally standard Neural Networks, there are three possible functions for σ [7]:

- The sign function:

$$\sigma(x) = \text{sgn}(x) \tag{2}$$

- The threshold function:

$$\sigma(x) = \mathbb{1}_{[x>0]} = \max\{x, 0\} \tag{3}$$

- The sigmoid function:

$$\sigma(x) = \frac{1}{1 + \exp(-x)} \tag{4}$$

These functions seek to emulate the activation of a biological unit. As a result as a reflection to the biological analogue, they are labelled as the activating functions of the neurons. The input of such neurons are described as the weighted sum over all of the neurons that output into it, with weights prescribed by w. Let us assume without the loss of generality that in our construction of the mathematical graph G, that it is arranged in discrete layers, such that the union of the set of layered vertices regenerates V, that is;

$$V = \bigcup_{i=0}^{I} V_i \tag{5}$$

where i marks the i-th layer of our generated Network. with the 0-th layer is labelled the input layer, for which input vectors a are fed. Under such an organised structure, the j-th neuron in the i-th, we can define the input into the $i+1$ layer of the j-th neuron as;

$$x_{i+1,j}(a) = \sum_{r \text{ s.t } v_{i,r}, v_{i+1,j} \in E} w((v_{i,r}, v_{i+1,j}))o_{i,r}(a) \tag{6}$$

where the output of the $i+1$-th layer and j-th neuron is $o_{i+1,j} = \sigma(x_{i+1,j}(a))$. Once such a neural network has been constructed we can obtain a hypothesis class $\mathcal{H}$ for which learning can be conducted, for which the weights w can be successfully learnt through backpropogation.

2.3 Rectified Linear Units

Rectified Linear Units or ReLu, is an activation function that seeks to threshold values at 0. First introduced in [8], the activation seeks to model the biological synaptic response due to its one-sided nature in lieu of the anti-symmetric responses given by sigmoidal activation such as tanh. Furthermore the linear units exhibits remarkable efficacy in implementation resulting from the sparse activation's and heightened gradient propagation. A unit is given by:

$$\sigma(x) = \mathbb{1}_{[x>0]} = \max\{x, 0\} \tag{7}$$

and is the standard for most activations within modern network architectures. Weights initialisation needs to be carefully considered otherwise there is a risk of impeding learning. In this assignment, we applied the He [9] initialisation for RELU.

2.4 Stochastic Gradient Descent and Momentum

Stochastic Gradient Descent is a tool commonly employed within most modern machine learning methods. Similar to most Gradient Descent formulations, an objective function $J(\theta)$ is desired to be minimised with respect to the parameter set θ. In the case of most deep learning applications where the gradient of negative conditional log-likelihood of training is expressed as:

$$\nabla_\theta J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta L(x^{(i)}, y^{(i)}, \theta) \tag{8}$$

However in the case of large data-sets, the computation of this expectation becomes expensive proportional to the amount of samples m. As such, in lieu of calculating the expectation $E(J(\theta))$ by examining the cost and gradients of the entire training set, SGD based algorithms estimates such expectations by using the value of the gradients weighted from a sample within our training. Generally speaking, individual samples can yield large variances within the training set [7]. As such this leads to the usage of multiple samples in a "mini-batch" to decrease such variances. Formally speaking let us define such a mini-batch to be the set of values $\mathbb{B} \subset X$ such that:

$$\mathbb{B} = \{x^1, \dots x^{m'}\} \tag{9}$$

where $m' < m$ is a preset value across training. Then we can present the estimate of the gradient as:

$$g = \frac{1}{m'} \nabla_\theta \sum_{i=1}^{m'} L(x^{(i)}, y^{(i)}, \theta) \tag{10}$$

then the gradient descent algorithm is updated through successive iterations as follows:

$$\theta := \theta - \eta g \tag{11}$$

for some learning rate η.

2.5 Mini-Batch Training

Typically, gradient descent algorithms that uses the entire training dataset are called batch gradient methods due to the fact that such algorithms process the examples within the set in a singular batch. On the other hand, mini-batch methods such as those employed in the formulation of Stochastic Gradient Descent, takes a $\mathbb{B} \subset X$ without loss of generality, such that computation times are rapidly improved [10]. There are multiple advantages of taking such batches $\mathbb{B}$, but primarily this reduces the large variance [11] that is derived from taking a singular sample as previously discussed. Mini-batch trainings can generate such $\mathbb{B}$ at each timestep t through sampling the training set X both with and without replacement. Due to the fact that replacement generates bias within training, most deep learning applications chose to generate batches without replacement as a means to mitigate such biases. Furthermore, the size of such batches are decided through consideration of multiple factors. One such factor is the fact that the larger the batch size, the more accurate the gradient estimations tend to be. This can be seen in the limit of equation (10) as $m' \to m$. On the other-hand, the efficiency of most modern multi-threaded computer architectures in calculating

extremely small batches tend to be extremely high, leading to under-utilisation of hardware [12]. In order to generate an optimal batch for convergence these factors must be tested in any architecture of multi-batching.

2.6 Momentum

Despite the efficacy of Mini-Batch Stochastic Gradient descent in optimisation, the rate of convergence may be slower than desired in highly convex functions with steep ravines along local optimal points. As such the method of momentum, developed by Polyak [13] is an optimisation that provides high convergences when the desired function possess noisy gradients. Developed through consideration of an analogue to dynamical quantities such as velocities and momenta, the optimisation considers past gradients in the form of a moving average, updating a velocity vector v_t at each timestep t:

$$v_t = \gamma v_{t-1} + \eta g \tag{12}$$

for which;

$$\theta = \theta - v_t \tag{13}$$

where $\gamma < 1$ is some fraction that simulates a computational analog to dynamical resistance, g is the estimated gradient through mini-batching SGD. In doing so, this reduces the oscillatory patterns that may be observed in noisy gradients. This leads to a faster rate of convergence as the momentum term increase if pointing in the direction of the previously calculated gradient and penalises if pointing otherwise.

2.7 ADAM optimisation

ADAM or Adative Moment Estimation is a refinement of the momentum method described above. Proposed in [14], the optimisation utilises the first moment, coupled with the second moment of the gradients within the learning. Unlike the aforementioned momentum method, the learning rate is adaptive at each iteration of the algorithmic process. We reproduce the ADAM iteration steps as in [14]:

$$m^{(i+1)} := \beta_1 m^{(i)} + (1-\beta_1) g^{(i)} L \tag{14}$$

$$v^{(i+1)} := \beta_2 v^{(i)} + (1-\beta_2)(g^{(i)} L)^2 \tag{15}$$

$$\hat{m} = \frac{m^{(i+1)}}{1-\beta_1^{(i+1)}} \tag{16}$$

$$\hat{v} = \frac{v^{(i+1)}}{1-\beta_2^{(i+1)}} \tag{17}$$

$$w^{(i+1)} := w^{(i)} - \eta \frac{\hat{m}}{\sqrt{\hat{v}} + \epsilon} \tag{18}$$

where $m^{(i)}$ is the first moment estimation with bias correction $\hat{m}$ at the i-th step and $v^{(i)}$ the second moment estimation with bias correction $\hat{v}$ at hte i-th step. At the end of each iteration the weight w is returned. By utilising such an optimisation method, the rate of convergence is much higher than regular SGD methods, along with the ability to escape "ravine" structures within the gradient estimation with relative ease. Given its efficacy in yielding high computational performances, most "out-of-the-box" neural network architectures utilises ADAM as its gradient descent optimiser [15]

2.8 Batch Normalisation

As the depth of Deep learning architectures increase, the difficulty of convergence within descent algorithms correspondingly increase because the data distribution changes at each layer when previous layer's parameters are refined. This causes training to slow down as it requires a lower learning rate to compensate. In order to address such issue, a variety of regularisation are available. One such regularisation is Batch Normalisation (BN), as proposed by [16]. The author also proposed that through the use of BN, the need to use Dropout is eliminated and the learning rate can be set higher. During each iterative pass of training, BN continuously updates the intermediate output of mini-batch SGD architectures by normalisation through calculating batch $\mathbb{B}$ means and variances. Within each activation of layer V_i, BN aims to transform the input x as follows:

$$f(x) = \alpha \left(\frac{x - \hat{\mu}}{\hat{\sigma}} \right) + \beta \tag{19}$$

where $\hat{\mu}$ and $\hat{\sigma}$ are the mean and standard deviation estimates of batch $\mathbb{B}$ at a certain timestep without loss of generality, and α and β are scaling and offset coefficients. This effectively rescales the input data for each layer to be normally distributed, which allows for the justification of large learning rates without divergence. Furthermore, noise generated from the estimation of mean and variance within each batch act as a means of regularisation that prevent overfitting from large network depths

2.9 Dropout Layers

As a network becomes deeper, the neuron starts to learn the specific details of the training features, thus become overly complex and overfit through co-adaptation. Another method of regularisation is the implementation of dropout throughout the network. First introduced by Srivastava [17], dropout works through the training of the ensemble consisting of sub-networks that are generated through the random removal of certain non-output nodes within a certain network. In doing so, this ensemble are prevented from co-adapting when some of the neurons are removed. The author also suggested dropout rate of 20% and 50% at the input and hidden layers respectively is found to be optimal.

2.10 Softmax and Cross Entropy Loss

The final layer of our architected neural network is the commonly known soft-max layer. This relies on outputting a probability distribution based on a process called soft-max regression which is analogous to logistic regression

for n classes that is predicted. We will not present the theoretical aspects of soft-max regression but rather provide its activation and cost function in order to discuss its back propagation error and its effect in training the weights w of the neural network. The activation is given by;

$$y_i = \sigma(z)_i = \frac{e^{z_i}}{\sum_j^C e^{s_j}} \tag{20}$$

Where C is the number of classes and j the j-th class. Then the corresponding cross-entropy is;

$$E = -\sum_{i=1}^{C} t_i \log(y_i) \tag{21}$$

2.11 Weight Decay Regularisation

In order to further regularise learning, it is possible to employ a process called weight decay. By encouraging weight to be small at each update step of the Neural Network algorithm it increases the generalisation power of the model [18]. At each update of our gradient descent, the weights are multiplied by a factor $\lambda < 1$ such that learning does not overfit the parameters. Generally speaking, the effect of weight decay is understood to be that of implementing $L2$ regularisation to the cost function that we are minimising, however the exactness of such a claim may not be exactly true. [19] examines this effect of adding weight decay to a deep training network, and suggests that despite exhibiting similar regularisation effects, this may not be the case.

3 Experiment and Results

3.1 Module Implementation and Experimental Setup

Given that the aim of this project is to implement a multi-layer neural network without the usage of "out-of-the-box" packages, we seek to generate each aforementioned learning module through the definition of custom functions. This was done using *python* which is well known for its versatility in generating programs without a remarkable hit in performance as is expected from high-level programming languages. Utilising its extensive *numpy* library to perform mathematical operations pertaining to linear algebra, we generate a base multi-layer neural network to perform classification on an unknown data set. This includes generating the entire neural network structure encompassing the appropriate initialisation for weights, bias, layers, activation function, forward and backward propagation. Furthermore functions required for evaluation and model refinement such as cross validation, training and hyper-parameter tuning are created. Finally through the implementation of the modules discussed in our methodology in our base Neural Networks, we are able to examine its effect on model accuracy in a heuristic process through 3-fold cross validation. In particular the effects of Dropout was examined in comparison to Batch Normalisation, along with both being applied. Mini-batching SGD with momentum was further bench-marked with conventional ADAM optimiser [14] as a means of model evaluation. We generate a model structure composed of an input layer along with 2 hidden layers followed by that of a softmax output. A diagrammatic representation of such a proposed architecture is reproduced in Figure 1.

3.2 Specifications and Runtimes

Each successive version of the Network architecture trained with modules added on are tested on an Intel Core i5-7360U Mac OSX enviroment and an Intel Core i7-8565U, Arch-Linux environment. Models were trained for 60 epochs due to the time consuming nature of running deep layer architectures without using graphic cards. The experiments were repeated for 10 times to ensure reliability of results. Each iteration was timed and the average run-times were taken.

The run-times for each Network version is provided in the table below [1].

Table 1: Total run-times of applied regularisation methods for each optimisation method (with 3-fold cross validations).

Regularisation Method	SGD	SGD MO	Adam
Dropout (0.1,0.3,0.5)	6.4min	7.1min	8.3min
Batch Normalisation (BN)	2.4min	2.5min	3.0min
Dropout And BN	3.5min	3.7min	4.6min

3.3 Dropout

The effects of adding Dropout as a means of regularisation on the Network has a distinct impact on the results of training the Network. It can be see in Figure 2, Figure 3 and Figure 4 that despite dropout reducing the training and validation accuracy, with higher levels of dropout yielding poor accuracy, the lower levels of dropout (≈ 0.1) exhibit robust training accuracy similar to that of the original network trained.

[0] Analysis of Results shall be given in the Discussion section

[1] Despite the high run time required for ADAM optimisation, the rate of convergences for these architectures are remarkably different. See section 4

Table 2: Average Training and Validation Accuracy with Dropout rate

Rate	Accuracy	SGD	SGD MO	Adam
0.1	Train	77.91%	88.81%	95.56%
0.1	Val	77.70%	88.40%	93.95%
0.3	Train	76.51%	87.70%	94.02%
0.3	Val	76.27%	87.39%	92.70%
0.5	Train	73.76%	86.51%	92.59%
0.5	Val	73.52%	86.22%	91.48%

Table 3: Average Training Loss with Dropout rate

Rate	SGD	SGD MO	Adam
0.1	0.514	0.292	0.144
0.3	0.597	0.339	0.193
0.5	0.736	0.388	0.246

3.4 Batch Normalisation

Training and Validation accuracy of Batch Normalisation conducted exhibited remarkably high results as seen in Table 4 at the end of a 60 epoch training session. The accuracy for each epoch are plotted appropriately in Figure 5, Figure 6 and Figure 7 with three optimiser respectively.

Table 4: Average Training and Validation Accuracy with Batch Normalisation

Accuracy	SGD	SGD MO	Adam
Train	77.74%	89.72%	96.88%
Val	77.50%	89.31%	94.81%

Table 5: Average Training Loss with Batch Normalisation

SGD	SGD MO	Adam
0.719	0.313	0.117

3.5 Dropout and Batch Normalisation

Given that both Dropout and Batch Normalisation are independent of each other, it is possible to examine their effects in one singular run of the experiment. The average accuracy of the combined regularisation are given in Table 6, with accuracy per epoch given in Figure 8, Figure 9 and Figure 10 with three optimiser respectively.

Table 6: Average Training and Validation Accuracy with BN and Dropout

Accuracy	SGD	SGD MO	Adam
Train	68.78%	84.09%	89.40%
Val	68.31%	83.92%	88.80%

Table 7: Average Training Loss with BN and Dropout

SGD	SGD MO	Adam
1.546	0.695	0.439

3.6 Optimiser Comparison

The training and validation accuracies of Stochastic Gradient Descent, SGD with momentum and ADAM optimisation methods were plotted, with implemented dropouts. This allows us to compare the relative performance of each optimiser with dropouts further elucidating the power of each optimiser in showing heightened convergence rates despite lowered information gain through generalisation of dropout. The training and validation accuracy of SGD are presented in Figure 2, Figure 5 and Figure 8, momentum in Figure 6, Figure 9 and Figure 3, and ADAM in Figure 7, Figure 10 and Figure 4. We can see that the rates of convergence are higher with each moment added in our algorithm, with momentum reaching steeper accuracy saturation, and ADAM exhibiting even faster saturation.

3.7 Hyper-parameter Tuning and Classification of Test Set

Through analysing the differing classification accuracies on validation and training data in utilising the above regularisations and optimisation we present the following as consideration for further hyper-parameter tuning:

Despite the high run-time of ADAM operations, its rate of convergence is seen to be much higher than that of both SGD and the method of momentum. From this it can be seen that applying early stopping will increase the speed of our algorithm. However since we aim to draw a comparison with that of other optimisers, we still run ADAM in parallel time of a full 60 epochs. Furthermore by comparing the loss charts several trends can be examined;

1. Batch Normalisation can be regarded as a constraint on the input samples to each layer. Due to its ability to rescale data, larger learning rates can be chosen, along with reducing the model's co-dependence on dropout as well as long as careful weight initialisation is implemented.
2. Utilising this understanding, we can generate bounds for a grid-search in tuning the learning rates. If the learning rate is too small (such as 10^{-5}), the loss will converge to minima very slowly. Comparatively, if the learning rate is too large (such as 10^{-2}), loss growth will undergo explosive behaviour with current loss per iteration being up to three times as high as that of previous iterations.
3. As such we limit grid search properties to:
$$\eta \in [10^{-4}, 10^{-3}] \tag{22}$$

Considering such facts that we have explored, a variety of cross-validating procedures and parameter grid-search was

conducted, with the following hyper parameters (Table 8) decided to be optimal for training our model and applying classification to the given unknown test-set. From our consideration of the above results we choose to utilise a customised ADAM optimiser with mini-batched normalisation in order for training. Dropout was not considered due to a variety of issues for which we will discuss in 4.

Utilising such hyper-parameters and training over a period of 60 epochs (due to large computational demands and for relative comparison to that of SGD and momentum), the final training and validation accuracy achieved was seen to be 97.917% and 95.296% respectively. The same model configuration was use to generate an output file consisting of the predicted labels of the given unknown dataset. The class distribution of these labels are given in Figure 11.

Table 8: Selected Hyper-parameters for test set classification

Hyper-parameter	**Value**
Learning Rate	$4.93056310 \times 10^{-3}$
Learning Decay	0.95
Weight Decay	1×10^{-6}

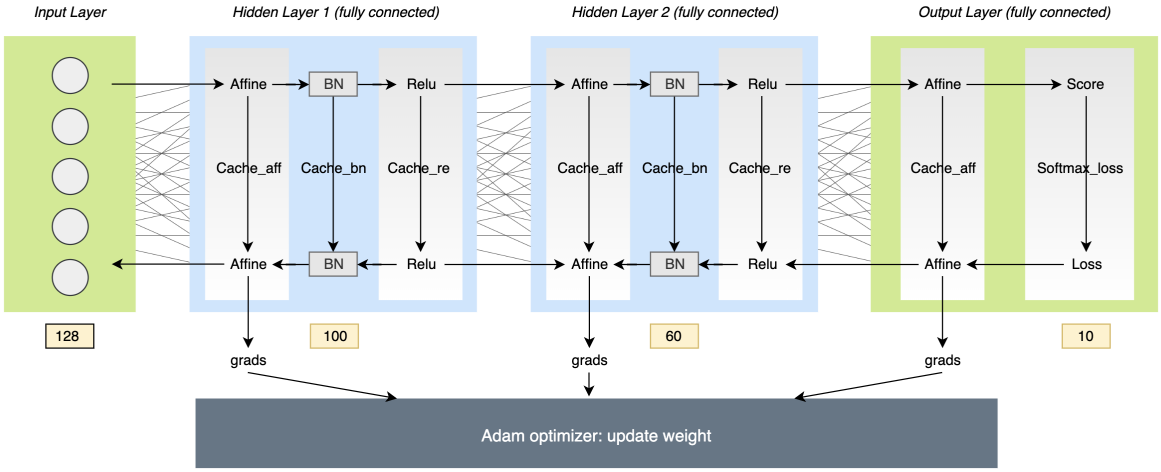

Figure 1: Network Structure of Final training model for classification of Unknown test set.

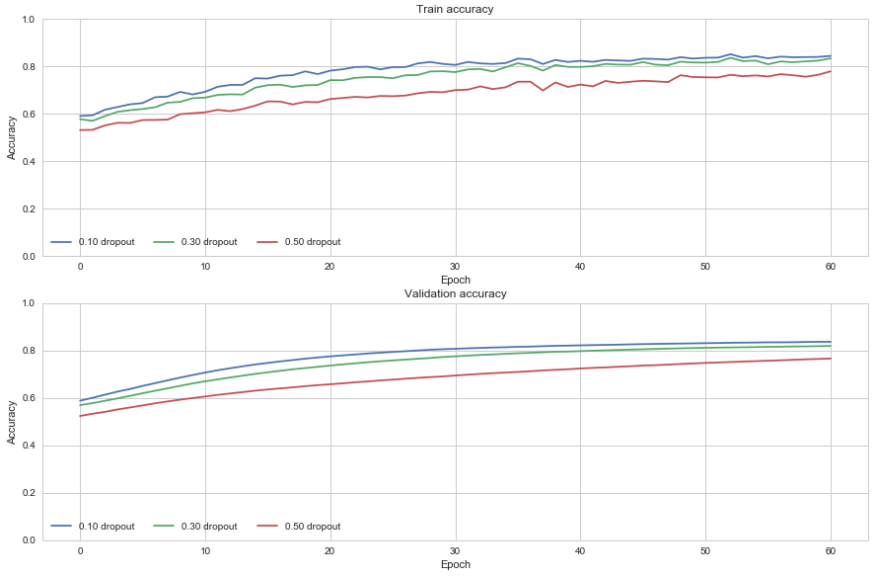

Figure 2: Training and Validation accuracies as a function of Epoch's trained conducted with Stochastic Gradient Descent optimiser. Dropouts from 0.1, 0.3, 0.5 are labelled accordingly.

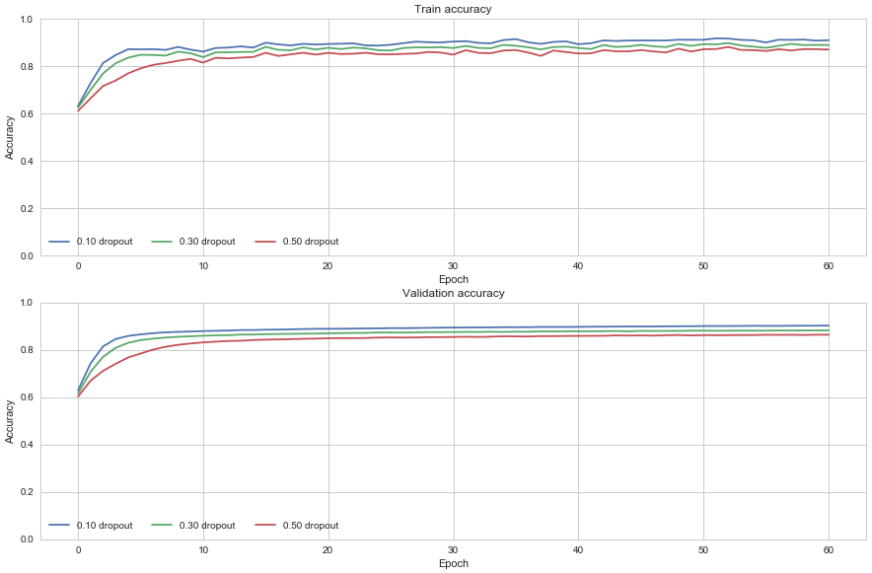

Figure 3: Training and Validation accuracies as a function of Epoch's trained conducted with Stochastic Gradient Descent Momentum optimiser. Dropouts from 0.1, 0.3, 0.5 are labelled accordingly.

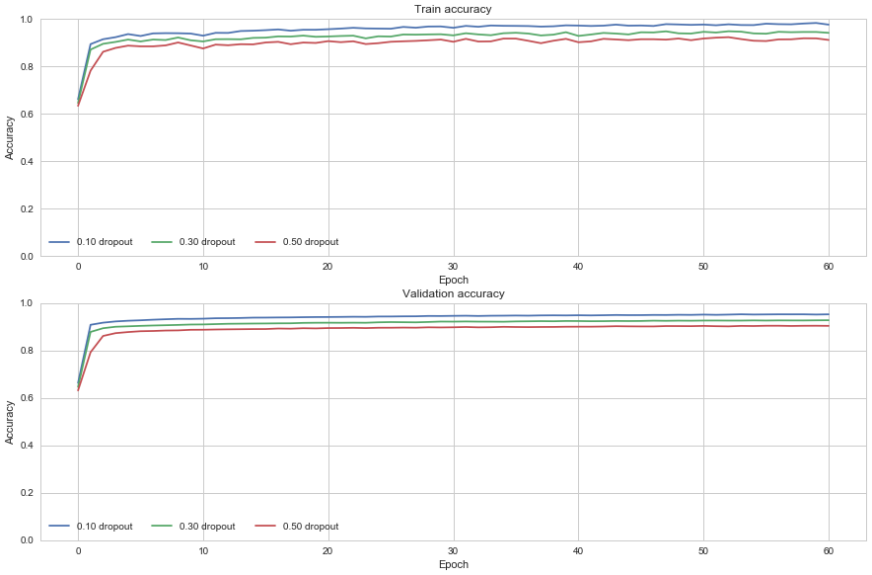

Figure 4: Training and Validation accuracies as a function of Epoch's trained conducted with Adam optimiser. Dropouts from 0.1,0.3 and 0.5 are labelled accordingly

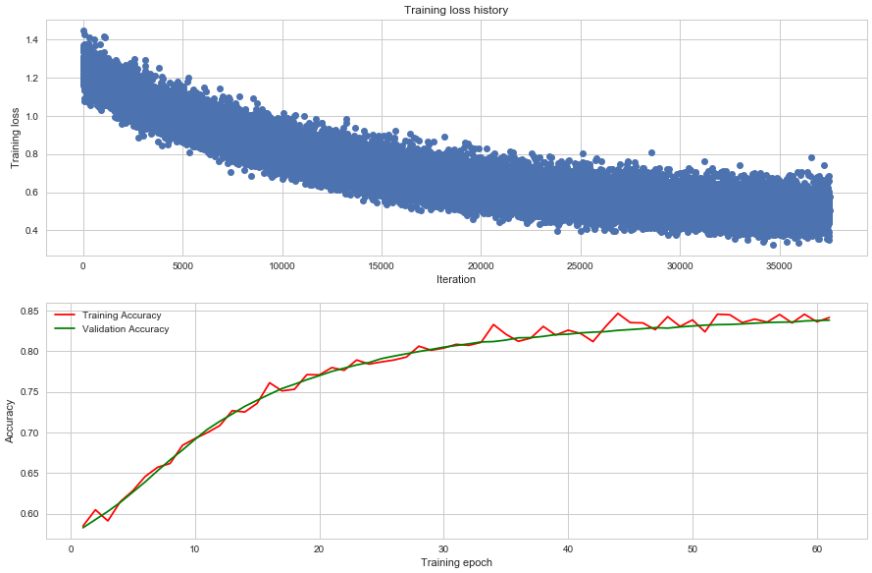

Figure 5: Training and Validation Accuracies conducted with Batch normalisation (bottom)with Stochastic Gradient Descent optimiser, Training Loss Distribution as Epoch size increases (Top)

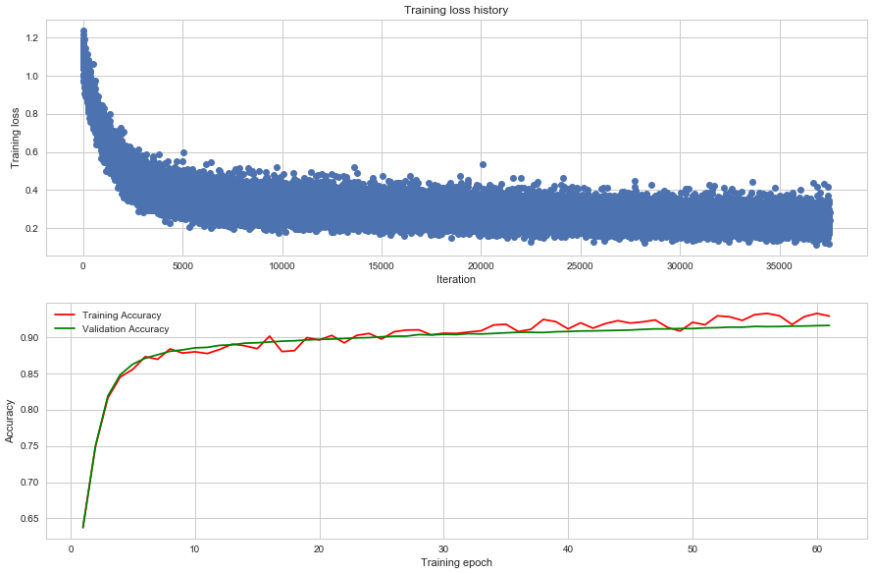

Figure 6: Training and Validation Accuracies conducted with Batch normalisation (bottom)with Stochastic Gradient Descent Momentum optimiser, Training Loss Distribution as Epoch size increases (Top)

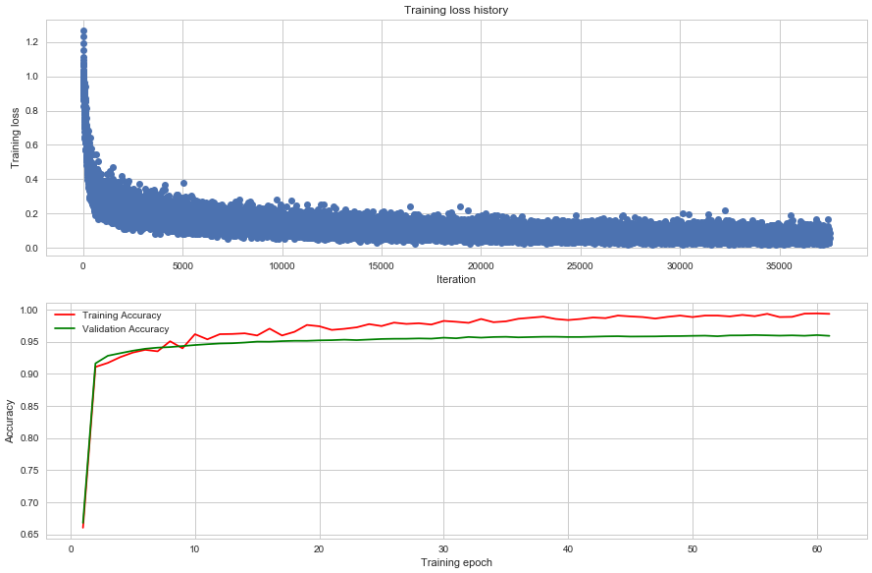

Figure 7: Training and Validation Accuracies conducted with Batch normalisation (bottom)with Adam optimiser, Training Loss Distribution as Epoch size increases (Top)

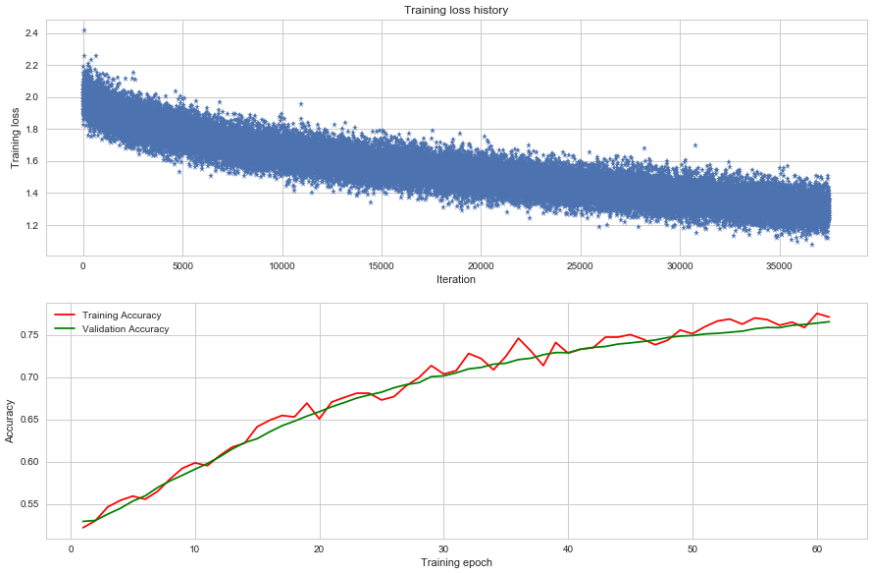

Figure 8: Training and Validation accuracies conducted with Batch normalisation and Dropout combined (bottom)with Stochastic Gradient Descent optimiser, Training Loss Distribution as Epoch size increases (Top)

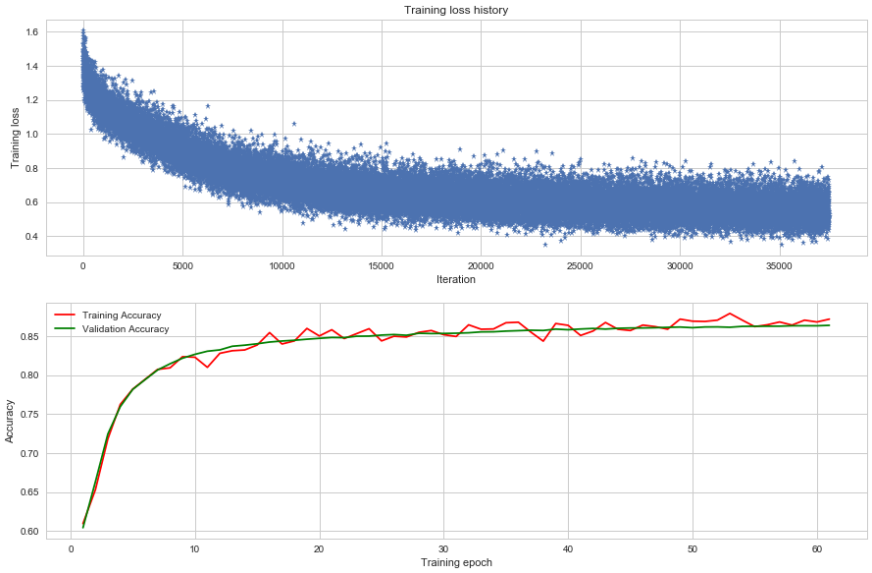

Figure 9: Training and Validation accuracies conducted with Batch normalisation and Dropout combined (bottom)with Stochastic Gradient Descent Momentum optimiser, Training Loss Distribution as Epoch size increases (Top)

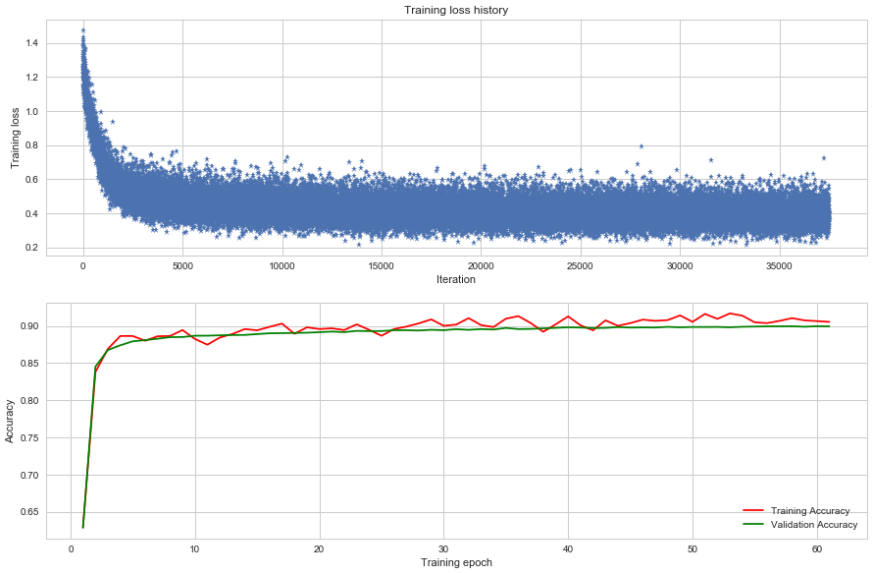

Figure 10: Training and Validation accuracies conducted with Batch normalisation and Dropout combined (bottom)with Adam optimiser, Training Loss Distribution as Epoch size increases (Top)

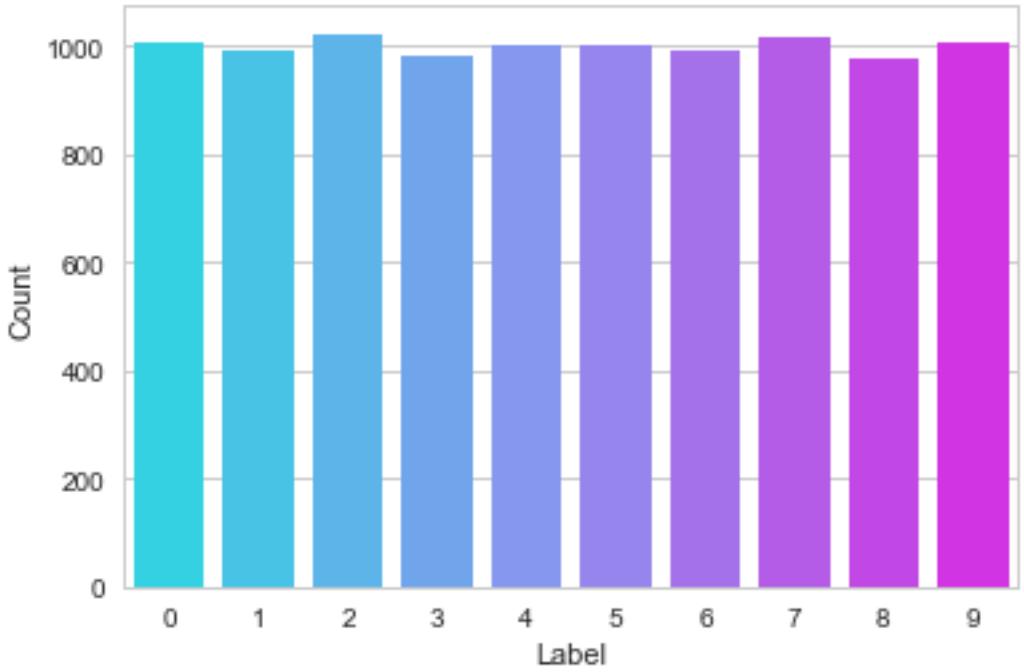

Figure 11: Distribution of Predicted Class labels on Unknown test set.

4 Analysis and Discussion

To evaluate and benchmark our given models in a practical environment, we use the accuracy metric to determine Network model performances. Accuracy reflects the percentage of correctly labelled observations to an applied dataset. In the case of our study, we are given an unknown testset in which we will predict the labels using the model that was vigorously trained and tuned. To infer the performance, we instead examine the training and validation predictions accuracies as a measure of the model performance.

It can be seen that in our presentation of Dropout, heightened levels of dropout performed such as that of 0.3 and 0.5 the training and validation accuracies decreases sharply. However we consider that it may be due to the amount of epoch's that have been chosen to be too small to examine the behaviour of these dropout rates at longer epoch times (> 60). This is not as remarkable as we would otherwise expect as comparable literature such as [20] exhibits lowered training accuracies. However, due to the aforementioned regularisation effect discussed in section 2.9, we expect that the model generalises rather well to testing situations to prevent possible over-fitting concerns that may arise form our given training set. On the other hand, Batch Normalisation procedure implemented overall generates a high accuracy of roughly 94.81% suggesting a general suitability in creating a high performing Neural Network structure, as it also allows for the adoption of a highly aggressive learning rate. As seen in the run-time of the architecture consisting of Batch Normalisation implementations, the run-times are lower than that without. This suggests an agreement with that of theoretical suggestions from literature [16]. Given such success from this task, the results from the combination of Dropout and Batch Normalisation yields a surprising reduction in prediction accuracy on trained validation data. The sub 90% accuracies in comparison to that of both Dropout and Batch normalisation is remarkable, however comparable studies into this phenomenon [21] suggest that it may be due to variance shifting effects that generates a biased prediction to occur. As such in consideration of a final testing classification for our given unknown test set, we preclude such an event from occurring through removing Dropout from our implementation and opt instead for the faster convergence rates of Batch Normalisation.

Although regularisations are an important part of generating high performing Network architectures with model generalisation, the optimisation strategy that should be chosen are also particularly important. It can be seen from the above presentation of the run-times along with the prediction accuracies despite the choice of SGD (both with and without the method of momentum) taking shorter time periods than that of ADAM based optimisation, the convergence saturation occurs much earlier in ADAM close to the second epoch. We suggest that perhaps this is due to the required calculation of the second moment on top of that of the first as seen in SGD momentum, which increases the algorithmic complexity. Furthermore we can see that the magnitude of fluctuations are comparatively lower for SGD momentum than that of purely SGD, along with ADAM exhibiting less fluctuations than that of momentum. We can attribute this in the same manner of discussion between the usage of SGD without its momentum implementation. In generating a moment (First order for momentum and

second for ADAM), the convergence rate of the optimisation is remarkably higher, without remarkable loss of accuracy.

Despite the evident success of generating accurate predictions with relative high computation performance, the run-times of our generated functions in comparison to that of "out-of-the-box" implementations are excessively high. This suggests inefficiencies that arise from resource utilisation without our computation tasks. It should be noted within our study, the implementation uses hardware demands of a CPU, notably that of the Intel Core i7-8565U and which despite its powerful computational power, pales in comparison to the utilisation of graphic cards such as that in [22] . As such in future studies conducted, it is recommended that Graphic card or multi-threading be implemented to improve resource assignment in increasing architecture performance. Also in generating a full dataset classification, without the need for comparison to other optimisation methods, early stopping at convergence saturation is recommended for practical use. Furthermore as noted above, the disharmony between dropout and Batch normalisation could be addressed with the use of more 'stable-variance' dropout formulations such as that described in [21], which may allow for a higher performing model that that constructed in our given implementation. We also suggest that models be run for higher periods of time that that of 60 epochs due to examining whether high dropout rates converge at higher levels of epoch training.

5 Conclusion

It can be seen that the ability of Neural Networks to create classifications in an accurate manner is varied depending on the modules implemented within the Network architecture. Such performances are heavily impacted by the regularisation methods and optimisation that have been used. It can be clearly delineated that the choice of an Adaptive Learning Optimiser such as ADAM along with normalisation with mini-batching generates a high-performing structure within the use of a multi-core CPU architecture has allowed us to achieved training/validation accuracy at 97.917% / 95.296% and for us to generate a classification to an unknown dataset with high confidence. Through such an exploration into the differing methods along with a variety of aforementioned improvements, we are able to suggest and design higher performing Network structures for future network designs as a means of classification.

References

[1] Dan C. Cireşan, Ueli Meier, Jonathan Masci, Luca M. Gambardella, and Jürgen Schmidhuber. Flexible, high performance convolutional neural networks for image classification. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence - Volume Volume Two*, IJCAI'11, pages 1237–1242. AAAI Press, 2011.

[2] Dan C. Ciresan, Ueli Meier, and Jürgen Schmidhuber. Multi-column deep neural networks for image classification. *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pages 3642–3649, 2012.

[3] and. Multilabel neural networks with applications to functional genomics and text categorization. *IEEE Transactions on Knowledge and Data Engineering*, 18(10):1338–1351, Oct 2006.

[4] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357, 2002.

[5] Bikesh Kumar Singh, Raipur, Kesari Verma, N. I. T. Raipur, and Aniruddha Santosh Thoke. Investigations on impact of feature normalization techniques on classifier's performance in breast tumor classification. 2015.

[6] Christian Szegedy, Alexander Toshev, and Dumitru Erhan. Deep neural networks for object detection. In *Advances in neural information processing systems*, pages 2553–2561, 2013.

[7] S. Shalev-Shwartz and S. Ben-David. *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press, 2014.

[8] Richard HR Hahnloser, Rahul Sarpeshkar, Misha A Mahowald, Rodney J Douglas, and H Sebastian Seung. Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit. *Nature*, 405(6789):947, 2000.

[9] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[10] Jakub Konečnỳ, Jie Liu, Peter Richtárik, and Martin Takáč. Mini-batch semi-stochastic gradient descent in the proximal setting. *IEEE Journal of Selected Topics in Signal Processing*, 10(2):242–255, 2016.

[11] Rie Johnson and Tong Zhang. Accelerating stochastic gradient descent using predictive variance reduction. In *Advances in neural information processing systems*, pages 315–323, 2013.

[12] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. Adaptive Computation and Machine Learning series. MIT Press, 2016.

[13] Boris Polyak. Some methods of speeding up the convergence of iteration methods. *Ussr Computational Mathematics and Mathematical Physics*, 4:1–17, 12 1964.

[14] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[15] Alfredo Canziani, Adam Paszke, and Eugenio Culurciello. An analysis of deep neural network models for practical applications. *arXiv preprint arXiv:1605.07678*, 2016.

[16] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *ICML*, 2015.

[17] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014.

[18] Anders Krogh and John A Hertz. A simple weight decay can improve generalization. In *Advances in neural information processing systems*, pages 950–957, 1992.

[19] Guodong Zhang, Chaoqi Wang, Bowen Xu, and Roger Grosse. Three mechanisms of weight decay regularization. *arXiv preprint arXiv:1810.12281*, 2018.

[20] Nitish Srivastava. Improving neural networks with dropout. *University of Toronto*, 182:566, 2013.

[21] Xiang Li, Shuo Chen, Xiaolin Hu, and Jian Yang. Understanding the disharmony between dropout and batch normalization by variance shift. *arXiv preprint arXiv:1801.05134*, 2018.

[22] Zhongwen Luo, Hongzhi Liu, and Xincai Wu. Artificial neural network computation on graphic process unit. In *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, volume 1, pages 622–626. IEEE, 2005.

A Code Instructions

Prerequisite : python3, hd5py, numpy, matplotlib, seaborn, random, pandas.

1. Unzip file
2. Edit the *superMLP.py* file by changing input and output paths for line 42 and 43 to that of local user.
3. Open up a python run-time.
4. Type *'import superMLP'* and wait for code to run.(Look at the Results.ipynb file for example)

Alternatively;

1. Steps 1 to 2 as before,
2. Open a terminal or command-line emulator
3. Run the program:

```
> python superMLP.py
```