Ten-class classification using multi-layer neural network

Chen Chen 480458339
Mingxuan Li 470325230

29th April 2019

Abstract

This project numerically illustrates the effects of various state-of-the-art modules in a feed forward neural network by performing a ten-class classification task. Section 1 briefly reviews relevant work in the field of neural network. Section 2 describes the modules applied in our neural network and their theoretical support. Numerical results from one-way sensitivity analysis, stated in Section 3, agree with the theories. While our most accurate model has a 90.0% cross validation accuracy, our best model achieves a cross validation accuracy of 89.9% in 2.3 minutes. Future work involves implementing the proposed backward propagation algorithm using the NVIDIA GPU based package `Cuda`.

Contents

1 Introduction

This study builds a multi-layer neural network with cross-entropy loss to perform a ten-class classification task. We implement various state-of-the-art neural network modules to improve the test accuracy and convergence rate classifying the given dataset. The applied modules consist of Xavier initialisation [1], rectified linear unit (ReLU) [2], dropout [3], momentum method [4], weight decay [5], mini-batch training [6], and batch normalisation [7]. We numerically validates the performance of these modules by the ten-class classification problem. With these modules, our model accurately classifies the cross validation (CV) dataset with an accuracy of 89.9% within 2.3 minutes. To illustrate the effects of each module, our numerical experiment gradually turns the modules off one by one, and then explore the numerical consequences by comparing them with the original benchmark model.

Neural networks were inspired by the biological neural networks which constitute human brains [8]. The concept of perception was originally introduced as a probabilistic model for image recongition [9]. The invention of back-propagation allowed perceptron to be extended to multi-layer neural networks [10]. As the first significant application, LeCun et al. [11] applied backward propagation to train computers to read human handwriting. The recent advances in computer hardware made the backward propagation algorithm feasible for training large and deep neural networks. Consequently, neural networks became more prevalent. Recently, researchers in this field discovered many empirical and heuristic techniques to improve the performance of neural networks in accuracy and convergence rate. These techniques include the

modules applied in our benchmark model introduced in the first paragraph.

Neural networks are now widely applicable in many fields such as finance [12, 13], facial recognition [14, 15], and more recently natural language processing [16, 17]. It is irreplaceable by other machine learning models because of its distinguishing features.

For example, neural network can accurately classify complex and highly non-linear dataset. The structure of nonlinearity behind artificial intelligence problems is usually difficult to formulate. Unlike classic statistical models, neural networks are able to autonomously learn the nonlinearity from complex data and extract latent features [18]. To increase the complexity of the hypothesis class (i.e. the number of latent features), we can conveniently raise either the number of layers or the number of hidden neurons per layer. With a suitable size of the hypothesis class, the neural network model is capable to extract latent features from complex datasets.

Moreover, the training process of neural network can efficiently utilise the computational power from modern multi-core CPUs and GPUs. The most computationally intensive part of training a neural network is to compute matrix multiplications and Hadamard product for large matrices. Modern languages like `Python` perform these matrix operations very effectively with highly vectorised code [19], through either CPU or GPU. Apart from vectorised calculations, packages like `Numpy` is also optimised to avoid copying data in memory, and to minimise the operation counts while performing matrices operations [19]. These features of modern language allow the training neural network becomes more efficient.

The rest of this report firstly introduces the loss function, the modules applied and optimisation techniques. This is followed by the experiment setup and detailed simulation results. Finally, we discuss importance of all the modules applied and how our neural network model benefits from them.

2 Methods

A neural network is a machine learning model that uses a network of functions in multiple layers to understand and translate data inputs into a desired outputs [18]. In classification problems, it learns from extracting the features of many labelled training samples that are usually hidden, and compares its predicted outputs against the given labels. For each iteration, the loss function is propagated back through the network, to adjust the weights in each layer. This process is repeated until a desired accuracy is achieved. The training algorithm is often referred to as a backward propagation algorithm, which is useful for training multi-layer feed forward neural networks.

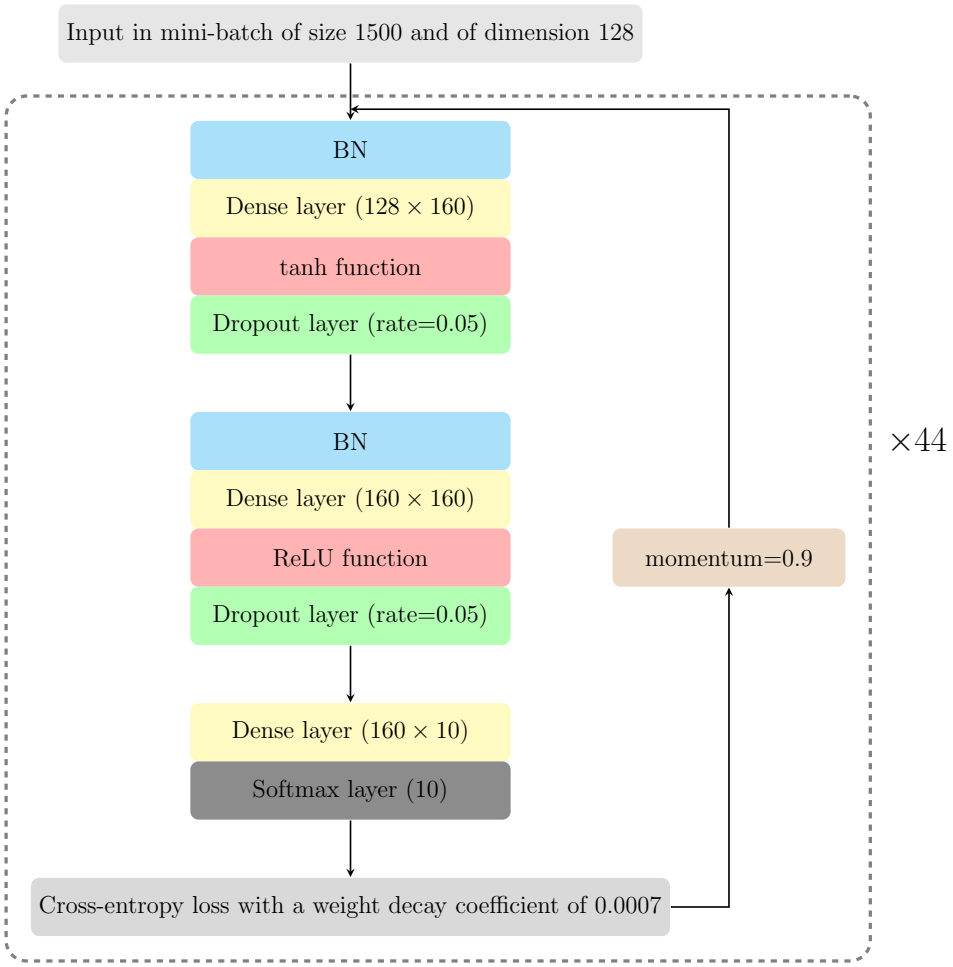

Figure 1: A diagram illustrates the architecture of our benchmark neural network model. The weights of the densely connected layers are initialised by a uniform distribution proposed by Glorot and Bengio [1].

We build a fully connect feed forward neural network for the classification task with a cross-entropy loss and L2 weight decay regularisation. Our neural networks implements the Xavier uniform initialisation and three activation layers (Figure 1). This report uses $\vec{t}_i = (t_{i1}, t_{i2}, \ldots)$ to denote the inputs of the ith densely connected layer ($i = 1, 2, 3$) [1] , and uses $\vec{z}_i = (z_{i1}, z_{i2}, \ldots)$ to denote the outputs of the ith activation layer. We apply batch normalisation (BN) to normalise the input data from $\vec{t}_1$ to $\tilde{\vec{t}}_1$. This is followed by a nonlinear activation function. After the first hidden layer, our neural network applies

[1] We use ( ) to denote column vectors, and [ ] to denote row vectors.

a dropout layer, followed by an additional BN to the normalised outputs. We then apply an densely connected layer with a ReLU activation function, an additional dropout layer. The neural network finishes with a Softmax activation layer.

2.1 Cross validation score and early stopping prevents overfit

We partition our dataset into $50,000$ training samples and $10,000$ test samples. For each epoch, we first train our model using the training samples and then compute the CV accuracy using the test samples. After 200 epoch, we find the number of epoch `n_iter` corresponding to the maximum CV accuracy as our early stopping point. Using all the $60,000$ samples, we then train our model again `n_iter` epoches to obtain our final neural network model.

2.2 Cross-entropy with weight decay improves the model robustness

Define $\vec{y}_k := (y_{1k}, y_{2k}, \ldots, y_{10k})$ as the one hot encoding of the training label for the kth sample and $\vec{p}_k := (p_{1k}, p_{2k}, \ldots, p_{10k})$ as the corresponding predicted probabilities, where $\sum_{j=1}^{10} p_{jk} = 1$. We apply the cross-entropy loss to train our neural network

$$L(\vec{p}|\vec{y}) = -\sum_k \sum_{j=1}^{10} y_{jk} \log p_{jk}. \tag{1}$$

Although the structure of our neural network is relative simple, it still has too many parameters. For example, if we set the number of hidden nodes in each hidden layer to be 160, then the number of parameters to estimate in the weights W_1, W_2 and W_3 is $128 * 160 + 160 * 160 + 160 * 10 = 47,680$. The total number of parameters including biases b_i and BN parameters γ_i and β_i is even greater. As we only have $50,000$ training samples, allowing entries of the weights W_i to be arbitrary real number overfits the dataset. To encounter this ill-proposed problem, we penalise the complexity of our model by a L2 weight decay regularisation term as proposed by Krogh and Hertz [5] and Hoerl and Kennard [20]

$$L = -\sum_k \sum_{j=1}^{10} y_{jk} \log p_{jk} + \lambda \sum_{i=1}^{3} |W_i|^2, \tag{2}$$

where hyperparameter λ is a positive real number. In comparison with loss function (1), the regularised loss function (2) penalises the model to have large weight matrices W_i and hence reduces the model complexity.

The reason to choose cross-entropy loss lies in its derivative. Taking the derivative of loss function (2)

$$\frac{\partial L}{\partial z_{3j}} = \sum_k p_{jk} - y_{jk} + \lambda \frac{\partial}{\partial z_{3j}} \sum_{i=1}^{3} |W_i|^2 .$$

This derivative function has a key property that many other loss functions like mean square loss do not have—the larger the errors $\sum_k p_{jk} - y_{jk}$ is, the faster all the neurons will learn.

2.3 Xavier initialisation speeds up convergence

We initialise the weight matrix W_i for $i = 1, 2, 3$ using Monte Carlo method from a uniform distribution bounded by $\pm\sqrt{6/\left[\dim(z_i) + \dim(t_i)\right]}$ as suggest by Glorot and Bengio [1]. We use 0s as the initial condition for bias vector $\vec{b}$. As a result, the weight decay term helps to prevent the neural network from overfitting.

We also have experimented to use the normal distribution initialisation suggested by Glorot and Bengio [1], but it is outperformed in accuracy and speed. So, this initialisation method is not discussed here.

Xavier initialisation makes sure the initialised weights and biases are not too far away from the optimal weights and biases [1]. This is essential because a poorly initialised optimisation problem usually either ends up with a solution that is far away from global optimum, or even diverges.

2.4 Batch normalisation prevents internal covariance shift and smooths optimisation

We normalise the inputs t_{ij} for layers $i = 1, 2$. When training using mini-batch B as defined in Section 2.7.1, we use $\mu_B^{(j)}$ and $\sigma_B^{(j)^2}$ to denote the sample mean and sample variance for the mini-batch B. Here, each of the index j denotes one dimension of the input data. Moreover, we define γ_{ij} and β_{ij} $(i = 1, 2)$ as real parameters to be learnt by backward propagation. The BN layer iteratively normalise each dimensions j of the input batch as

$$\tilde{t}_{ij} = \gamma_{ij} \frac{t_{ij} - \mu_B^{(j)}}{\sqrt{\sigma_B^{(j)^2} + 10^{-8}}} + \beta_{ij} \text{ where } i = 1, 2. \tag{3}$$

When predicting test labels, we use the training means $\frac{1}{|B|}\sum_B \mu_B^{(j)}$ and training variances $\frac{1}{|B|}\sum_B \frac{|B|}{|B|-1}\sigma_B^{(j)^2}$ to normalise the test inputs. Our neural network

applies this BN layer to prevents internal covariance shift [7]. In addition, Santurkar et al. [21] find that BN makes the optimisation landscape considerably smoother.

2.5 Dropout performs model averaging

We apply dropout to the inputs of the 2nd and 3rd densely connected layers $\vec{t}_2$ and $\vec{t}_3$, as proposed by Srivastava et al. [22]. When training our neural network, the dropout layer randomly ignores neurons in vectors $\vec{t}_2$ and $\vec{t}_3$ with a given probability hyperparameter p. When predicting labels for CV samples and testing samples, we multiply weights W_i where layer index $i = 2, 3$ by hyperparameter p. This dropout layer is equivalent to model averaging, and forces our neural network to learn a more robust set of parameters that are useful in conjunction with many distinct random subsets of the neurons [3].

2.6 Activation functions map input to desired output range

Our neural network model has three densely connected layers. These three layers respectively implements a nonlinear activation function (tanh or sigmoid), a ReLU activation function, and a Softmax activation function.

2.6.1 Nonlinear activation

The first densely connected layer is chosen to be either of a hyperbolic function

$$\vec{z}_1(\tilde{\vec{t}}_1) = \tanh(W_1\tilde{\vec{t}}_1 + \vec{b}_1) \tag{4}$$

or a sigmoid function

$$\vec{z}_1(\tilde{\vec{t}})_1 = \vec{1} \Big/ \left[\vec{1} + \exp\left(W_1\tilde{\vec{t}}_1 + \vec{b}_1\right)\right] \tag{5}$$

where matrix W_1 is a 160×128 weighting matrix and vector $\vec{b}$ is an 160-dimensional bias vector. Here, the division / is an elementwise operator. This layer provides non-linearity to our neural network.

2.6.2 ReLU activation function

The second densely connected layer is the piece-wise linear function named as ReLU

$$\vec{z}_2(\tilde{\vec{t}}_2) = \max(0, W_2\tilde{\vec{t}}_2 + \vec{b}_2).$$

ReLU forces the output of the layer to be sparse. In contrast, the nonlinear activation functions (4) and (5) do not have this property. The sparseness makes the neural network more computationally efficient [23].

2.6.3 Softmax activation function

The Softmax activation functions takes a vector of real numbers as the input, and convert it into a categorical distribution as the output. Define matrix $\mathbf{1}$ to be a 10×10 matrix of ones so that pre-multiplying by matrix $\mathbf{1}$ is equivalent to summing up all rows. The activation function is the probability mass function of the categorical distribution

$$\vec{z}_3(\vec{t}_3) = \exp(W_3\vec{t}_3 + \vec{b}) \Big/ \left[\mathbf{1}\exp\left(W_3\vec{t}_3 + \vec{b}\right)\right], \tag{6}$$

which sums up to one. We assign the index of probability vector $\vec{z}_3(\vec{t}_3)$ corresponding to the largest probability as the classification result for the corresponding sample.

2.7 Stochastic gradient descent finds the optimal weights

We use gradient descent with mini-batch and momentum to iteratively find optimise our model.

2.7.1 Mini-batch improves the speed of computation

Traditional batch gradient descent uses all samples to estimate the gradient of the loss function. Estimating such a gradient using this way may be a very computationally expensive procedure. Alternatively, stochastic gradient descent uses only one sample to estimate the gradient. Although it economises the computational cost, the non-smooth nature makes the convergent much slower. We applied mini-batch as a compromise between the true gradient descent and the stochastic gradient descent methods. Instead of using either all samples or only using one sample to estimate the gradient, we use $1,500$ training samples (i.e. a "mini-batch" B) at each step. In comparison with stochastic gradient descent, mini-batch naturally takes advantage of the vectorisation nature of the `numpy` library and converges much faster [6].

2.7.2 Momentum accelerates rate of convergence

We use η to denote the learning rate, α to denote the momentum coefficient, $\Delta W_i^{(s)}$ to denote the update of weight W_i in the sth iteration, and ∇ to denote

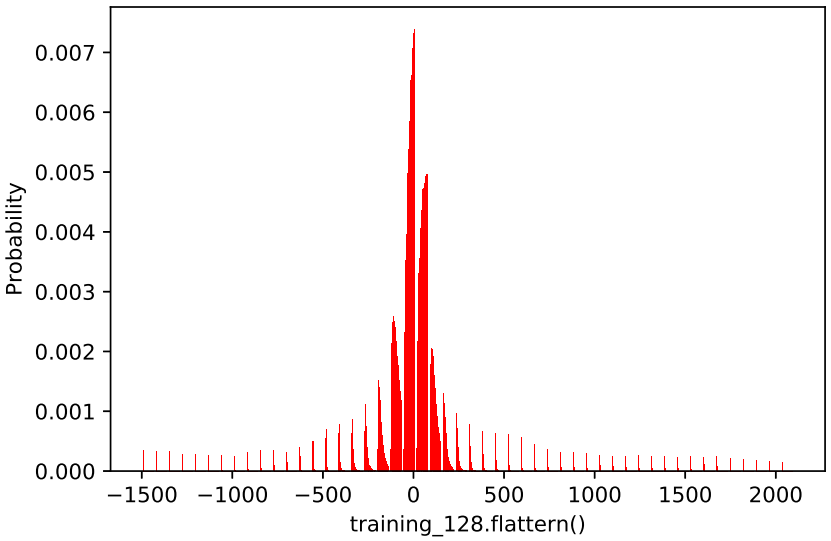

Figure 2: The frequency plot of our training input dataset. Most features of samples are within the range $(-300, 300)$. However, features can be as large as 2803.84 or as small as -2045.92.

the gradient operator. In the sth iteration, recall that the first order Taylor expansion of the loss function (2) informs to update the weight matrix W_i as $W_i^{(s+1)} := W_i^{(s)} - \eta \nabla L(W_i^{(s)})$, the gradient descent update with momentum proposed by Rumelhart et al. [4] is

$$\Delta W_i^{(s+1)} = \alpha \Delta W_i^{(s)} - \eta \nabla L(W_i^{(s)}) \text{ and } W_i^{(s+1)} = W_i^{(s)} + \Delta W_i^{(s+1)}.$$

To initialise this update, we set $\Delta W_i^{(0)} := 0$. This autoregressive update proposal smooths the wiggliness caused by gradient descent [4], and hence significantly accelerates the rate of convergence.

3 Experiments and results

3.1 Dataset

Our neural network is designed to classify $10,000$ unlabelled samples with 128 features and 10 different classes, using $60,000$ similar labelled samples. As described in Section 2.1, we split the labelled dataset into $50,000$ training samples and $10,000$ CV samples. The features in the dataset range from -2045.92 to 2803.84 (Figure 2).

Table 1: Hyperparameters for models which output outstanding CV accuracy. The benchmark model is our best model in terms of accuracy as well as speed. Hyperparameter sets 2–5 are either slightly less accurate or slower than our benchmark model. The row of running time is the time taken to train our neural network and make prediction with the computer detailed in Figure 3. The row of CV accuracy reports the classification accuracy of the CV samples, which includes the generalisation error and hence approximates the test accuracy. The row of initialisation states the supports of the uniform distribution in initialisation. The method of 'Xavier' is introduced in Section 2.3. The row 'Optimal epoch' states the number of epoch corresponding to the highest CV accuracy when training our model (Section 2.1).

Hyperparameters	Benchmark	Run 2	Run 3	Run 4	Run 5
Run time (minutes)	2.3	2.7	3.6	17.2	10.8
CV Accurarcy	89.9%	89.7%	89.8%	90.0%	89.3%
Initialisation	Xavier	Uniform $(-1, 1)$	Uniform $(-1, 1)$	Xavier	Xavier
Batch size	1500	1500	1500	1500	1500
Nodes per layer	160	150	150	900	160
Activation function	tanh	tanh	tanh	tanh	sigmoid
Weight decay rate	0.0007	0.0007	0.0007	0.0007	0.0007
Momentum rate	0.9	0.9	0.92	0.9	0.9
dropout rate	0.05	0	0	0.5	0.05
Learning rate	0.11	0.05	0.05	0.11	0.11
Optimal epoch	44	54	66	158	282
BN	True	True	True	True	True

3.2 Use Numpy and Scipy for code vectorisation

We use `python` 3.6 to code the neural network model with a list of library dependencies specified in the file named `requirements.txt` file in our git repository. In particular, our model requires the scientific computing library `scipy 1.21`, and uses `numpy` to perform vectorised matrix algebra operations.

3.3 Multiprocessing speeds up hyperparameters tuning

Column 1 in Table 1 illustrates the hyperparameters of our Benchmark neural network, which completes the classification task with a CV accuracy of 89.9% within 2.3 minutes.

To demonstrate the effect of each module in the benchmark model, we perform one way sensitivity analysis by varying one selected module in each

Table 2: The accuracy of our benchmark model with one selected module either turned off or modified. The row of running time is the time taken to train our neural network and make predictions with the computer detailed in Figure 3. The row of CV accuracy reports the classification accuracy of the CV samples, which includes the generalisation error and hence approximates the test accuracy. The row of initialisation states the supports of the uniform distribution in initialisation. The method of 'Xavier' is introduced in Section 2.3. The row 'Optimal epoch' states the number of epoch corresponding to the highest CV accuracy when training our model (Section 2.1).

Hyperparameters	Drop out 50%	Momentum	Weight decay	Mini-batch	BN
Run time (mins)	0.5	8.6	0.8	7.6	4.5
CV Accurarcy	85.5%	89.4%	88.6%	87.5%	82.8%
Initialisation	Xavier	Xavier	Xavier	Xavier	Xavier
Batch size	1500	1500	1500	50000	1500
Nodes per layer	160	160	160	160	160
Activation function	tanh	tanh	tanh	tanh	tanh
Weight decay rate	0.0007	0.0007	0	0.0007	0.0007
Momentum rate	0.9	0	0.9	0.9	0.9
Dropout rate	0.5	0.05	0.05	0.05	0.05
Learning rate	0.11	0.11	0.11	0.11	0.11
Optimal epoch	9	198	16	177	80
BN	True	True	True	True	False

experiment. In the five experiments shown in Table 2, we turn up the drop out rate from 0.05 to 0.5, turn the momentum coefficient from 0.9 to 0, turn weight decay coefficient from 0.0007 to 0, replace mini-batch training by batch training, and train the model without BN, respectively.

We implement a `shell` script to run all these tests in parallel in a high performance computer, whose hardware specification is shown in Figure 3. The `shell` script trains our model with various hyperparameters from different `json` files in parallel. Running multiple experiments in parallel allows us to fully utilise our computing resources, and tune hyperparameters efficiently. Although the speed for each task has been dropped by 30%, this parallel setting dramatically reduces the total running time required to finish the ten experiments detailed in Tables 1 and 2. For example, fitting our neural networks with these ten different sets of hyperparameters in parallel takes 17.2 minutes, which actually equals the time required by the largest model in Table 1. However, fitting the ten neural networks in sequence takes a total of 44 minutes, which doubles the total time required in the parallel setting.

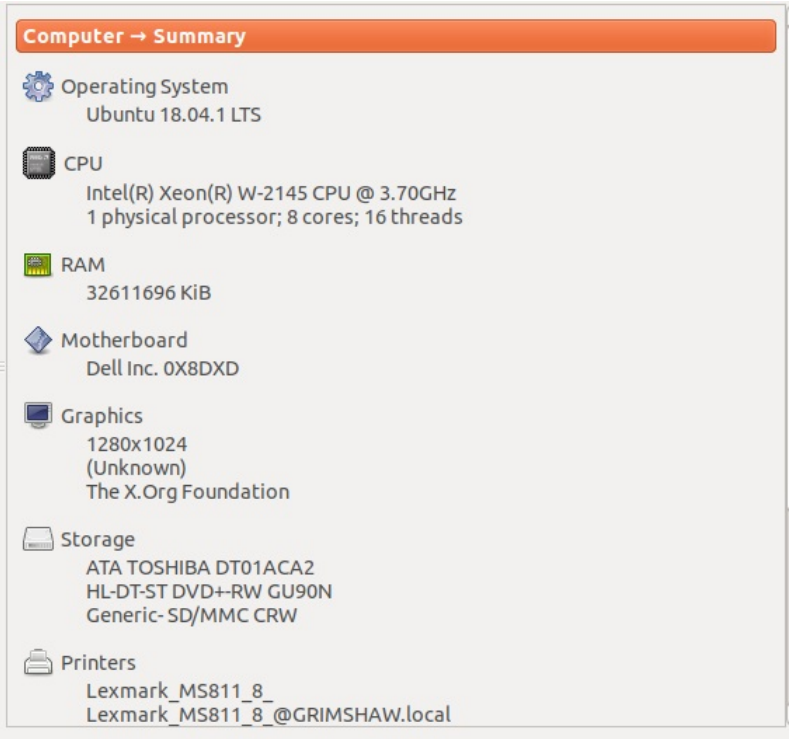

Figure 3: The hardware specification of our high performance computer.

3.4 Experiments Results

3.4.1 Early stopping decides optimal number of epochs

For the benchmark model listed in Table 1, early stopping detailed in Section 2.1 finds 44 epochs give the best CV accuracy, which is 89.9% (Figure 4). As a result, we choose 44 epochs as our early stopping point for the benchmark model.

3.4.2 Optimal hyperparameters

In terms of accuracy and training speed, our best neural network model is the 'benchmark' model detailed in Table 1. Although the CV accuracy of the benchmark model is 0.1% lower than the 4th hyperparameter who has 900 nodes in each hidden densely connected layers, it is 8 times faster. As a result, it is selected as our benchmark model. Table 2 and Figure 4 illustrate the impact of drop out, momentum, weight decay, mini-batch and BN, respectively, in the Benchmark model. With the optimal hyperparameters, backward propagation iteratively finds the weight matrices w_1, w_2, w_3 as

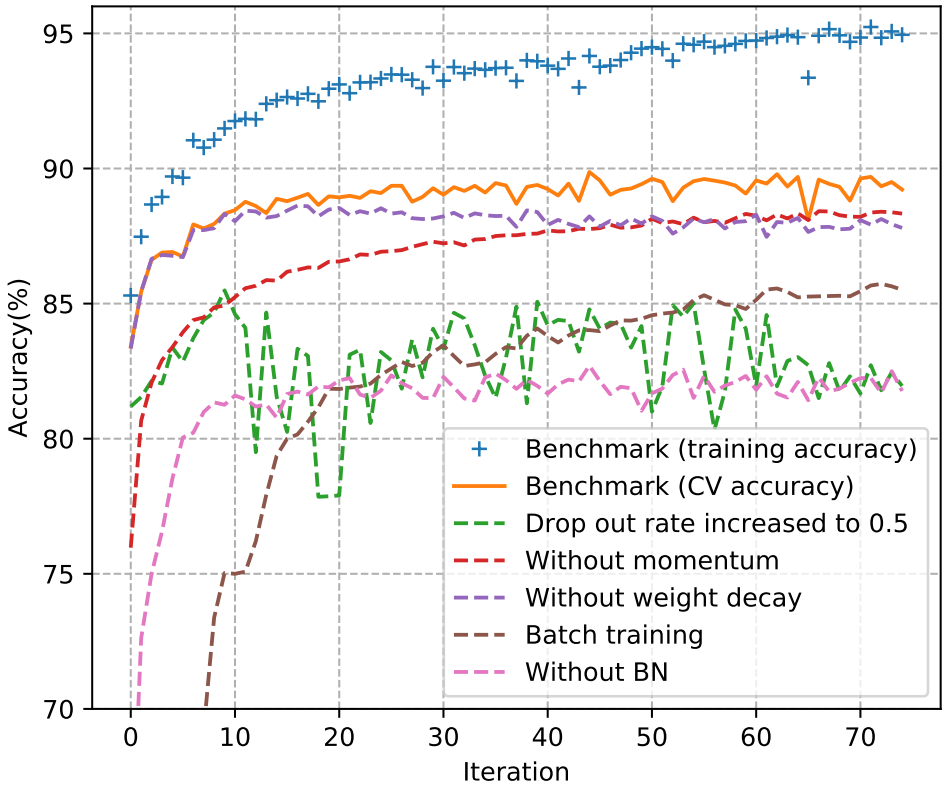

Figure 4: The cross plots the training accuracy of our benchmark model, and the solids line plots the CV accuracy. The dashed lines plots the CV accuracy of modified benchmark models where exact one hyperparameter is modified from the benchmark model.

visualised in Figure 5.

3.4.3 Dropout helps in large neural networks

In the benchmark model, turning up the drop out rate from 5% to 50% not only makes the model less accurate (Table 2), but also makes thw CV accuracy volatile (Figure 4). However, with the 50% drop out rate, the largest neural network model with 900 nodes per hidden layer turns out to be our most accurate model, which has a CV accuracy of 90% (Table 1).

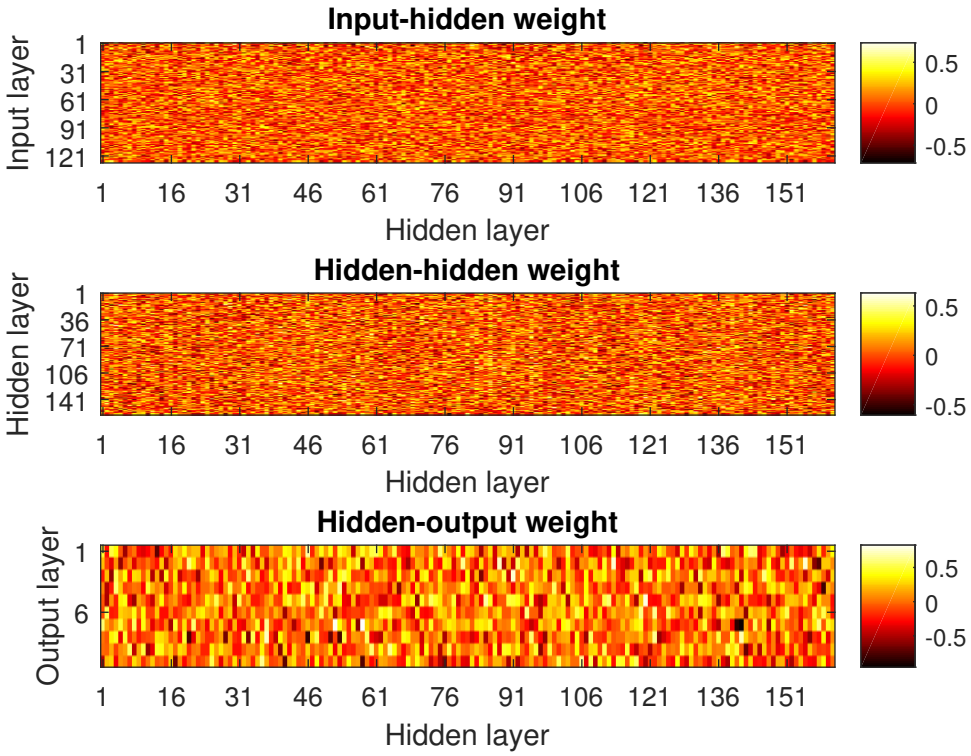

Figure 5: A visualisation of the optimal weight matrices for our benchmark model.

3.4.4 Momentum and mini-batch speed up the convergence

When turning off the momentum, column 2 in Table 2 shows that the time required to train the neural network model suddenly increases from 2.3 minutes (Benchmark model) to 8.6 minutes. This is also demonstrated in Figure 4. Moreover, comparing column 2 and 3 in Table 1, it seems that training sometimes is slower if the momentum coefficient is set to be too high.

In comparison with mini-batch training, batch training has a much longer training time (Column 4 in Table 2). The longer training time is caused by the slowness of large matrix multiplication in gradient descent.

3.4.5 Weight decay prevents over-fitting

Column 3 in Table 2 demonstrates that weight decay improves our model performance. As shown in Figure 4, the overfit when training neural network without weight decay drives the purple dashed line to decreases after epoch 14. In comparison, the orange solid line, which is the CV accuracy of

our benchmark model with weight decay, oscillates after epoch 44, without a trend of decline.

3.4.6 BN significantly improves the accuracy

In terms of accuracy, BN is the most important module in our neural network model. Turning off BN in our model decreases the CV accuracy from 89.9% to 82.8%.

3.4.7 Sigmoid is slower than tanh

We attempt to use the sigmoid activation function to replace the tanh function in the first densely connected layer. As shown at Column 5 in Table 1, this makes our model converges five times slower. Hence we choose tanh as the activation function in the benchmark model.

4 Discussion

Our simulation results (Table 1) illustrates that using tanh as the activation function in the first activation layer is a better choice than using sigmoid. This may be due to the fact that tanh has a steeper gradient in a large neighbourhood of zero. By tuning the hyperparameter, we seem to succeed to escape the 'vanishing gradient region', and hence the gradient in the neighbourhood of zero is more important than the gradient when inputs approach infinity. In the second activation layer, we implement ReLU because it only saturates in one direction. With ReLU, our neural network suffers less from the vanishing gradient problem [1].

Also, simulations results suggest that we need to be careful when tuning dropout and weight decay. In this simple classification problem, we have significantly more samples (60,000) compared with number of features 128. As a result, the generalisation error must be small [24], and a large dropout rate or a large L2 regularisation coefficient will over regularise our model [25]. Moreover, with the large sample size, we are confident that the CV accuracy of our benchmark model will be very close to our test accuracy that to be announced by the tutors.

Although Xavier initialisation only improves the accuracy by 0.1% (Columns 1 and 3 in Table 1), we find it makes the tuning of parameter much easier. This result may be due to Xavier initialisation helps the training process to escape the vanishing gradient region. In the meanwhile, the improved accuracy

might be explained by the hypothesis that Xavier initialisation moves our initial condition one step closer to the global optimum.

The results of this study show that mini-batch and momentum method considerably improve the speed of convergence. When applying Mini-batch, we take advantage of the code vectorisation feature of `numpy`, and hence improves the speed of computation. More importantly, by better estimating the gradient in each step, it improves the stability of our optimisation algorithm. Momentum acts as a damping term in optimisation and it smooths the oscillation in gradient descent [26]. Hence it accelerates the convergence of minimising the Cross-entropy loss.

Finally, our simulation results also confirms the outstanding performance of BN. BN provides the most significant accuracy improvement amongst all the modules. Rather than preventing internal covariants shift [7], it is now argued that BN plays an important role in smoothing the landscope of optimisation [21]. However, our experiment cannot verify which argument is more appropriate.

5 Conclusion

In conclusion, our numerical simulation supports the published results on various state-of-the-arts modules. We find BN is the most significant module to improve CV accuracy and predictive capability. In terms of speed, mini-batch training is the most important module follow by the momentum method. However, the size of the batch as well as the momentum coefficients need to be carefully tuned. Also, simulation results shows that dropout is useful for large neural networks where the number of parameters is large comparing with the number of samples. We use shell script to train neural network models with different hyperparameters in a parallel setting. Parallel computing fully utilises our computing resources and double the speed to train ten neural network models.

In recent years, NVIDIA released their Cuda package which parallelises matrix multiplications using GPU. Large and iterative matrix multiplications is the most computational intensive part when training neural network, This package improves the speed of matrix multiplication by a factor of > 20 [27]. As a computationally expensive procedure, a natural extension of this project is to implement Cuda to speed up the training speed of our neural network.

References

[1] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In Yee Whye Teh and Mike Titterington, editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy, 13–15 May 2010. PMLR. URL http://proceedings.mlr.press/v9/glorot10a.html.

[2] Richard Hahnloser, Rahul Sarpeshkar, Misha A. Mahowald, Rodney Douglas, and H Sebastian Seung. Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit. *Nature*, 405:947–51, 07 2000. doi:10.1038/35016072.

[3] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *CoRR*, abs/1207.0580, 2012. URL http://arxiv.org/abs/1207.0580.

[4] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–, October 1986. URL http://dx.doi.org/10.1038/323533a0.

[5] Anders Krogh and John A. Hertz. A simple weight decay can improve generalization. In J. E. Moody, S. J. Hanson, and R. P. Lippmann, editors, *Advances in Neural Information Processing Systems 4*, pages 950–957. Morgan-Kaufmann, 1992. URL http://papers.nips.cc/paper/563-a-simple-weight-decay-can-improve-generalization.pdf.

[6] Priya Goyal, Piotr Dollár, Ross B. Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch SGD: training imagenet in 1 hour. *CoRR*, abs/1706.02677, 2017. URL http://arxiv.org/abs/1706.02677.

[7] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 448–456, Lille, France, 07–09 Jul 2015. PMLR. URL http://proceedings.mlr.press/v37/ioffe15.html.

[8] S. N. Londhe and M. C. Deo. Artificial neural networks for wave propagation. *Journal of Coastal Research*, 20(4):1061–1069, 2004. ISSN 07490208, 15515036. URL http://www.jstor.org/stable/4299364.

[9] Frank Rosenblatt. The perceptron: A probabilistic model for information

storage and organization in the brain [j]. *Psychol. Review*, 65:386 – 408, 12 1958. doi:10.1037/h0042519.

[10] P. J. Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, Oct 1990. ISSN 0018-9219. doi:10.1109/5.58337.

[11] Yann LeCun, Bernhard E. Boser, John S. Denker, Donnie Henderson, R. E. Howard, Wayne E. Hubbard, and Lawrence D. Jackel. Handwritten digit recognition with a back-propagation network. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems 2*, pages 396–404. Morgan-Kaufmann, 1990. URL http://papers.nips.cc/paper/293-handwritten-digit-recognition-with-a-back-propagation-network.pdf.

[12] E. Michael Azoff. *Neural Network Time Series Forecasting of Financial Markets*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1994. ISBN 0471943568.

[13] Iebeling Kaastra and Milton Boyd. Designing a neural network for forecasting financial and economic time series. *Neurocomputing*, 10(3): 215 – 236, 1996. ISSN 0925-2312. doi:https://doi.org/10.1016/0925-2312(95)00039-9. URL http://www.sciencedirect.com/science/article/pii/0925231295000399. Financial Applications, Part II.

[14] S. Lawrence, C. L. Giles, , and A. D. Back. Face recognition: a convolutional neural-network approach. *IEEE Transactions on Neural Networks*, 8(1):98–113, Jan 1997. ISSN 1045-9227. doi:10.1109/72.554195.

[15] H. A. Rowley, S. Baluja, and T. Kanade. Neural network-based face detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(1):23–38, Jan 1998. ISSN 0162-8828. doi:10.1109/34.655647.

[16] Ronan Collobert and Jason Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th International Conference on Machine Learning*, ICML '08, pages 160–167, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-205-4. doi:10.1145/1390156.1390177. URL http://doi.acm.org/10.1145/1390156.1390177.

[17] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. Natural language processing (almost) from scratch. *J. Mach. Learn. Res.*, 12:2493–2537, November 2011. ISSN 1532-4435. URL http://dl.acm.org/citation.cfm?id=1953048.2078186.

[18] Christopher M. Bishop. *Pattern Recognition and Machine Learning*

(Information Science and Statistics). Springer-Verlag, Berlin, Heidelberg, 2006. ISBN 0387310738.

[19] S. van der Walt, S. C. Colbert, and G. Varoquaux. The numpy array: A structure for efficient numerical computation. *Computing in Science Engineering*, 13(2):22–30, March 2011. ISSN 1521-9615. doi:10.1109/MCSE.2011.37.

[20] Arthur E. Hoerl and Robert W. Kennard. Ridge regression: Biased estimation for nonorthogonal problems. *Technometrics*, 12(1):55–67, 1970. doi:10.1080/00401706.1970.10488634. URL https://www.tandfonline.com/doi/abs/10.1080/00401706.1970.10488634.

[21] Shibani Santurkar, Dimitris Tsipras, Andrew Ilyas, and Aleksander Madry. How does batch normalization help optimization? In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 2483–2493. Curran Associates, Inc., 2018. URL http://papers.nips.cc/paper/7515-how-does-batch-normalization-help-optimization.pdf.

[22] Nitish Srivastava, Geoffrey E. Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1): 1929–1958, 2014. URL http://dl.acm.org/citation.cfm?id=2670313.

[23] Roi Livni, Shai Shalev-Shwartz, and Ohad Shamir. On the computational efficiency of training neural networks. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 855–863. Curran Associates, Inc., 2014. URL http://papers.nips.cc/paper/5267-on-the-computational-efficiency-of-training-neural-networks.pdf.

[24] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An introduction to statistical learning*, volume 112. Springer, 2013.

[25] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. Springer New York Inc., New York, NY, USA, 2001.

[26] Gabriel Goh. Why momentum really works. *Distill*, 2017. doi:10.23915/distill.00006. URL http://distill.pub/2017/momentum.

[27] D. Strigl, K. Kofler, and S. Podlipnig. Performance and scalability of gpu-based convolutional neural networks. In *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, pages 317–324, Feb 2010. doi:10.1109/PDP.2010.43.