COMP5329 Assignment1

Tutor:

Group Members	SID	Unikey
Cheng Lu	480286592	chlu8208
Haeri Min	480401391	hmin5434
Albert Huang	530215563	qhua0468

1 Introduction

1.1 Aim

The aim of this assignment is to gain insights into the effectiveness of different strategies in building a Multilayer Neural Network (MNN) classifier using fundamental scientific computing packages like NumPy and SciPy. The primary objective is to create our own MNN classifier that predicts a multi-class classification task. This report provides a comprehensive illustration of the implemented methods, and experimental results, with a discussion on factors affecting model performance.

1.2 Background & Purpose

Deep learning has revolutionised various fields by enabling computational models to learn representations of data with multiple levels of abstraction. Its ability of automatic learning from raw data, without manual feature extraction, has led to major advancements in speech recognition, image classification, and drug discovery.[1]

This study is important as it provides a valuable learning opportunity for students with a deeper understanding of MNN by creating modules without using Deep Learning frameworks such as PyTorch, Tensorflow, Caffe, and KERAS. In addition, it allows us to understand the impact of different modules on model performance. It is desired that the completion of this assignment will contribute to a deeper understanding of deep learning practices and theories for all group members.

2 Methods

2.1 Pre-Processing

2.1.1 Class imbalance

Given that the dataset consists of 10 classes, we checked if there is balanced data across all labels. If more data is present in a certain class, this will potentially cause bias in the classification model as high accuracy will be achieved in a class with more samples relative to the minority class. Therefore, we checked for class imbalance and discovered that samples are evenly distributed across 10 labels with 5000 observations in each class (see Figure 1), hence there was no need for further pre-processing.

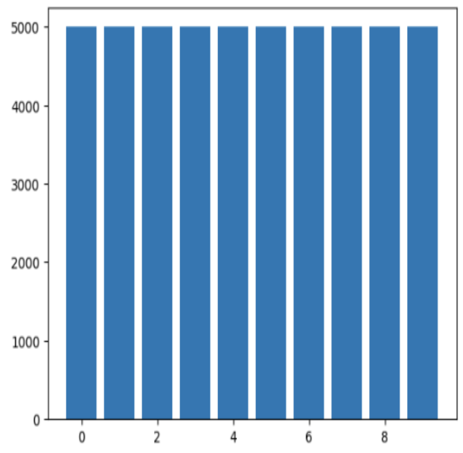

Figure 1: Summary of distribution of samples in each class in the training data

In addition to the class imbalance check, we examined if there were any non-numeric values reported in the training and test set features as non-numeric values have to be converted to missing values. No non-numeric values were detected in both training and test set features hence no further pre-processing was required.

2.1.2 Normalisation

Normalisation is an important data pre-processing technique as some features might have larger values than the others but this does not necessarily indicate that the variables are more important but it may be due to different scaling across the features. Moreover, machine learning algorithms that use gradient descent method as an optimisation technique require features to be normalised so that the gradient descent can converge to their minimum point much faster [2]. Looking at the raw training data, the features are scaled differently, hence we normalised the features in both training and test data to convert them to a common scale between 0 and 1. This was done by following Equation 1 before constructing a neural network.

$$X' = \frac{X - X_{min}}{X_{max} - X_{min}} \tag{1}$$

2.2 Principle of different modules

2.2.1 More than one hidden layer

Using multiple hidden layers in neural networks enhances the model's ability to learn complex features and patterns[3]. This architecture enables the network to capture hierarchical relationships in the data, extracting abstract features at deeper layers[4]. What need to mention is that there are studies shown that for general tasks, one or two hidden layers should be enough[5, 6, 7].

Training deep neural networks can encounter challenges like vanishing and exploding gradients[8]. These issues arise due to gradient propagation through numerous layers during backpropagation, hindering convergence[9]. Techniques such as careful weight initialization, proper activation functions, and batch normalization can mitigate these challenges and improve training stability[10, 11].

2.2.2 ReLU activation

Rectified Linear Unit (ReLU) is a widely-used activation function in neural networks for its simplicity and effectiveness. It introduces non-linearity by returning zero for negative inputs and leaving positive inputs unchanged[12]. The formula for ReLu function is shown as below.

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

This helps prevent the vanishing gradient problem, ensuring more stable and efficient training[9, 11]. Also, ReLU is computationally efficient which makes it suitable for training deep networks[12]. However, it may suffer from the "dying ReLU" problem, where neurons become inactive during training[13].

2.2.3 Weight decay

Weight decay, also known as L2 regularization, is a technique used to prevent overfitting in neural networks[14]. It involves adding a penalty term to the loss function that penalizes large weights[14].

Mathematically, weight decay is implemented by adding a regularization term to the loss function:

$$L = \text{original loss} + \lambda \sum_{i=1}^{N} w_i^2 \tag{3}$$

Where *L* is the new loss function, λ is the regularization parameter, *N* is the total number of weights in the network, and w_i^2 represents each weight. The regularization term $\lambda \sum_{i=1}^{N} w_i^2$'s effect is to penalizes large weights, which will discourage the network from fitting the training data too closely.

By including the regularization term in the loss function, weight decay encourages the network to learn simpler models with smaller weights, which tend to generalize better to unseen data[15, 16]. This helps prevent overfitting, where the model learns to memorize the training data rather than capturing its underlying patterns[14].

2.2.4 Momentum in SGD

Momentum is often used in stochastic gradient descent (SGD), helps speed up convergence and avoid getting stuck in local minima[17]. The momentum increases updates in dimensions with consistent gradient directions and reduces updates in dimensions with changing gradient directions, resulting in faster convergence and less oscillation[18]. The related equation of Momentum in SGD is shown below:

Eqn. (4) represent the current velocity, v_t, where γ is the momentum coefficient, v_{t-1} is the velocity from the previous step. η is the learning rate, and $\nabla_\theta J(\theta)$ represents the gradient of the loss function with respect to the parameter, θ.

$$v_t = \gamma_{t-1} + \eta \nabla_\theta J(\theta) \tag{4}$$

Eqn. (5) is reflective of the update of a parameter, where the current parameter is the previous parameter minus the current velocity.

$$\theta_t = \theta_{t-1} - v_t \tag{5}$$

2.2.5 Dropout

Dropout is a regularization technique used in neural networks that can prevent overfitting[19]. It randomly drops a certain proportion of neurons in a layer during training[19]. The equation of Dropout is shown below[20]:

$$y^{l+1} = \sum_M p(M) f((M * W) y^l) \approx f(\sum_M p(M)(M * W) y^l) = f(pWy^l) \tag{6}$$

y^{l+1} represents the output of the l+1-th layer. M represents a mask vector that indicates which neurons should be dropped out (0) or not (1). $p(M)$ calculates the probability distribution of the mask vector M. f is the activation function. W is the weight matrix. And y^l is the output of the l-th layer.

The process can be described as the equation computes the weighted sum of neuron outputs in the l+1-th layer, then performs a non-linear transformation using the activation function f. Dropout will then randomly drop some neurons by scaling their outputs with the mask vector M, this contributes to preventing overfitting and promoting robust learning.

2.2.6 Softmax and cross-entropy loss

In multi-class classification, the softmax function which is before the output layer assigns conditional probabilities to each of the classes[21]. It's often used in the last layer of neural networks for multi-class classification. Cross-entropy loss assesses the disparity between predicted probabilities (from softmax) and actual labels to tells us how well the predictions match the truth[22]. The equation for Softmax and cross-entropy loss are shown below[21]:

$$\text{Softmax}(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}} \tag{7}$$

$$\begin{aligned} CrossEntropy(t, z) &= -\sum_i t_i \log z_i \\ &= -\sum_i t_i \log t_i + \sum_i t_i \log t_i - \sum_i t_i \log z_i \\ &= -\sum_i t_i \log t_i + \sum_i t_i \log \frac{t_i}{z_i} \\ &= Entropy(t) + D_{KL}(t|z) \end{aligned} \tag{8}$$

The equation (7) represents the softmax function. It transforms the input vector z into a probability distribution. For each element z_i in the vector, it calculates the exponential of z_i divided by the sum of exponentials of all elements in the vector.

In the equation (8), the first term $-\sum_i t_i \log z_i$ calculates the negative log probability of the true classes under the predicted distribution. The second term $\sum_i t_i \log t_i$ calculates the entropy of the true distribution. The third one $-\sum_i t_i \log z_i$ represents the Kullback-Leibler divergence[23] between the true distribution and the predicted distribution.

2.2.7 Mini-batch training

Gradient descent is an optimisation technique that minimises a model's cost function. It iterates through training samples and computes model parameters in neural networks so that the loss between the expected and true values is minimised. The gradient descent formula is shown in the equation below.

$$\theta_j = \theta_j - \alpha \frac{d}{d\theta_j} J(\theta) \tag{9}$$

There are three different types of gradient descent: i) Stochastic Gradient Descent; ii) Batch Gradient Descent and iii) Mini-Batch Gradient Descent. Stochastic Gradient Descent computes gradient using only one training example whereas a batch gradient descent technique updates the gradient using the entire training data. Therefore, stochastic gradient descent leads to slower convergence and makes very noisy updates in the parameters whereas the batch gradient descent method makes smooth updates in the model parameters but requires more computational time to make a single update. To balance out the speedy convergence and smoothness of an update, a mini-batch gradient descent approach can be used which takes a subset of training data to update model parameters. Since it considers a subset and not the whole training data, it has a quicker convergence rate and is less noisy in the update.

Stochastic Gradient Descent computes the gradient for each training sample, X (refer to Eqn. (10)).

$$\theta = \theta - \eta \nabla_\theta J(\theta, X^{(i)}) \tag{10}$$

Batch Gradient Descent computes the gradient for the entire training dataset, that is, X from 1 to the end (refer to Eqn. (11)).

$$\theta = \theta - \eta \nabla_\theta J(\theta, X^{(1:end)}) \tag{11}$$

Mini-batch gradient descent computes the gradient for every mini-batch training samples, n (refer to Eqn. (12)).

$$\theta = \theta - \eta \nabla_\theta J(\theta, X^{(i:i+n)}) \tag{12}$$

Mini-batch gradient descent is an iterative process of computing gradients of loss with respect to the parameter, weight, and updating parameters with the gradients for each training batch. We first initialised weight, predicted a class using the current parameter and computed loss for a given batch (batch size of X was selected). After making a prediction for a given batch with the current weight, we then computed gradients of the loss (i.e. mean squared error) with respect to the weight and updated parameters with the gradients. The loss for each batch was stored and after performing gradient descent for all mini-batches, we averaged losses over all sample points to calculate the average loss.

2.2.8 Batch Normalisation

Normalisation is a pre-processing technique that transforms the values of features in a dataset to a scale range between [0,1] or [-1,1]. This ensures that all features contribute equally to the model and avoids the potential dominance of features with larger values [2] as the larger value does not necessarily indicate that a particular feature is relatively more important but mainly due to the difference in measurement scale (e.g.

income has relatively larger values than the years of education). It is applied to the input layer before passing the data to the network whereas batch normalisation is basically normalisation but inside the network, with hidden layers.

Batch normalisation has many advantages in practice including a reduction in training times, reduction in demand for regularisation, allows higher learning rates, and enables training with saturating non-linearities in deep networks e.g. sigmoid [20].

In terms of code execution in Python, we calculated the mean and variance of samples from a mini-batch and then normalised the samples following Eqn. (13). A small value, epsilon, was added to the variance to avoid zero variance. We also introduced two learnable parameters, gamma, and beta, where gamma is a scale parameter and beta is a shift parameter. These parameters are used so that the zero-mean and unit variance are not too hard for the network [24]. By applying normalisation using Eqn. (13), the new transformed training samples have zero mean and unit variance.

$$X_{new} = \frac{X - \mu}{\sqrt{\sigma^2 + \epsilon}} \tag{13}$$

2.2.9 Other advanced operations: Adam

Adam (Adaptive Moment Estimation) is an adaptive learning rate optimization algorithm commonly used in training neural networks. It combines the advantages of two other popular optimization techniques, Ada-Grad and RMSProp, by maintaining both per-parameter learning rates and an exponentially decaying average of past gradients[25].

The update rule for Adam involves computing the first and second moments of the gradients and using them to adaptively adjust the learning rates for each parameter. It is defined as follows[18]:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \tag{14}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \tag{15}$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \tag{16}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} \tag{17}$$

$$\theta_t = \theta_{t-1} - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \tag{18}$$

θ represents the model parameters. g_t is the gradient of the loss function with respect to the parameters. α is the learning rate. β_1 and β_2 are exponential decay rates for the moment estimates. m_t and v_t are relatively the estimations of the first and second moments of the gradients. $\hat{m}_t$ and $\hat{v}_t$ are bias-corrected estimates of the moments. ϵ is a small constant added to the denominator for numerical stability.

In our case, implementing Adam is expected to bring great performance boost.

2.3 Design of the best model

While designing our best model, we made the below decisions of attributions base on the information we have on dataset and principle of different modules.

1. **Cost-performance balance:** It is essential because it ensures that the model achieves optimal performance without requiring excessive computational resources. In practical terms, this assignment have 10 classifications and over 5,000 samples for each class which should be considered a medium to heavy task. Meanwhile under the requirements, all modules implemented using Numpy meaning that CPU is desired rather than on GPU. Only efficient modules should be considered and though a complex model could bring better performance, it would cost great computing resources.

2. **Modular Design:** The structure of our designed model should be flexible for modification and perform various kinds of experiments. Hence, Modular design is implemented in our codes and design. We create a class for each module and a class to assemble desired modules for running. And evaluation function is place separately after the above module. Using this method, our design of best model is more convenient to use or change and its readability are increased significantly, contributes to future updates.

3. **Alignment with task goal:** While designing our best model, we make sure each module we select and the combination of theses modules are aligned with task goal. This would help the model accurately capture relevant patterns and make informed decisions, leading to improved performance and usability. For example, for activation function, there are multiple choices like ReLU, LeakyReLU, Sigmoid, ELU, Tanh, Swish etc. We choose ReLu in the end base of its simplicity, efficiency and suitability in a classification task. Functions like Tanh (hyperbolic tangent) are disused due to its specialisations on sequence modeling, language modeling, and sentiment analysis.

With all the above considered, below is our design of best model:

Table 1: Summary of parameters used in our design of the best model.

	Module	Best Parameter
1	Number of neurons in each layer	120
2	Activation function	ReLu
3	Number of hidden layers	3
4	Momentum	None
6	Weight Decay	None
7	Mini-batch training	100
8	Normalization	True

For the rest of the experiments, we started with our design of best model as default, and through the process of experimenting, we try to find out what the practical best model would be.

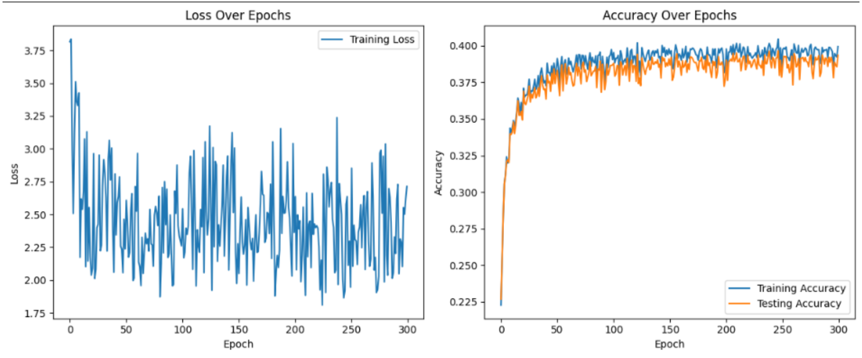

Figure 2: Standard Model Performance

2.3.1 Runtime comparison

The time taken to run the best model was 18 minutes and 21 seconds, whereas the standard model took 6 minutes 21 seconds.

3 Experiments and results

3.1 Model performance

As shown in Figure 3, training and testing accuracy increased with an increase in the number of epoch.

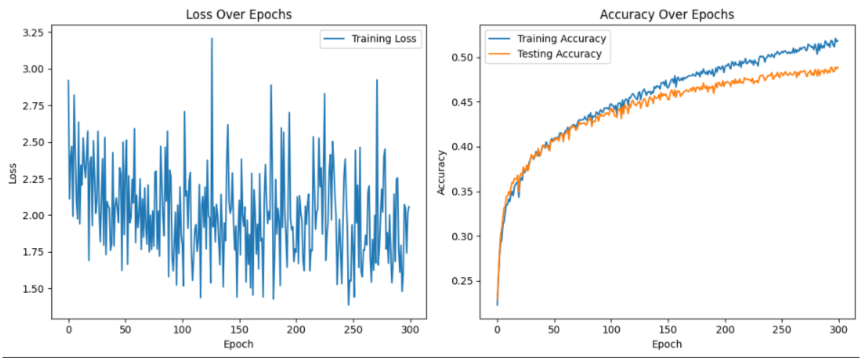

Figure 3: Best Model Performance Graph

We ran our deep learning neural network with different values of parameters and recorded corresponding test accuracy and training loss to evaluate and to understand how different parameter values impact the model performance (see Table 2).

Table 2: Summary of parameters used to create the best model.

Technique	Default Parameter	Test Accuracy	Training Loss
Number of neurons (60/120)	120	60:38.49 120:39.49	60:2.29 20:2.16
Activation Function (ReLu/Sigomid/Tanh)	ReLu	ReLu:39.73 Sigomid:38.70 Tanh:38.94	ReLu:2.17 Sigomid:1.93 Tanh:2.27
Hidden Layers (1/3)	1	1:39.25 3:2.12	1:48.88 3:2.17
Momentum (None/0.9)	None	None:39.79 0.9:8.28	None:2.13 0.9:8.10
Weight (None/0.2)	None	None:38.93 0.2: 15.57	None:2.25 0.2:2.28
Dropout Probability (0.5/0.8)	0.5	0.5:39.17 0.8:38.03	0.5:2.32 0.8:2.63
Mini-batch training (50/100)	100	50:37.77 100:39.73	50:2.29 100:2.11

3.2 Analysis

3.2.1 Number of neurons

There remains an unclear relationship between the number of neurons in the hidden layer and accuracy [26]. As discovered in Deng (2023) [26], increasing number of neurons does not necessarily increase the model performance however the accuracy will hover around at an optimal number of neuron and will likely to remain at a certain threshold [26]. This conforms with our results that increasing the number of neurons from 60 to 120 did not significantly increase the test accuracy (60: 38.49 vs 120: 39.49) nor the training loss (60: 2.29 vs 120: 2.16) as shown in Table 2.

3.2.2 Activation Functions

Three activation functions were applied to MNN: ReLu, Sigmoid, and Tanh. We observed that ReLu performed the best as the accuracy on the test set was 39.73, compared to sigmoid (test accuracy = 38.70) and tanh (test accuracy = 38.94). This is explained by the vanishing gradient descent problem in sigmoid and tanh which is alleviated in ReLu as the activation function ranges from 0 to infinity (i.e. the maximum threshold is infinity) [27].

3.2.3 Hidden Layers

In this analysis, both the test accuracy and training loss significantly improved when we used three hidden layers compared to one hidden layer. We expect more hidden layers will have better performance as they extract more features based on the theory. Compare to other modules, it has crucial impact and we believe it would be the fundamental for building the best model.

3.2.4 Momentum

Momentum is an aggregate of gradients that prevents sensitive movement when calculating gradient at every iteration. According to the results reported in Table 2, there was a significant difference between the model performance with and without using momentum (None: 39.79 vs 0.9: 8.28). As we do not have the information for the dataset, it is possible that the original data has outliers that still have effect even with normalisation.

3.2.5 Dropout Probability

Dropout randomly drops some neurons during training to prevent overfitting of the neural network. MNN performed the best with dropout = 0.5 which is consistent with the AlexNet structure that uses drop rate of 0.5 [28].

3.2.6 Mini-batch training

As explained in section 2.2.7, there are many advantages of mini-batch gradient descent over stochastic and batch gradient descent hence we tested with 50 and 100 mini-batch sizes to evaluate the effect of mini-batch size on model performance. There was a minimal difference in evaluation metrics between the mini-bath size of 50 and 100 (accuracy was 37.77 using size = 50 and 39.73 using size = 100). The results showed that splitting the training data into 100 batches improved performance.

3.3 Justification of your best model

As we want to minimise misclassification error and correctly predict the true label, we selected the best model based on the highest test accuracy and lowest training loss. In particular, using ReLU activation function maximised accuracy compared to tanh and sigmoid functions as it ranges from 0 to infinity, eliminating the vanishing gradient descent problem. Moreover, pre-processing the data using normalisation successfully enhanced the model performance as it scales parameters to a range between 0 and 1 so that no particular parameters are dominant in the neural network as larger value does not indicate greater significance in the network. Furthermore, using mini-batch size of 100 to perform gradient descent improved model performance, hence combination of these parameters were used to create the best model (as summarised in Table 1).

4 Discussion & Reflection

1. **Insufficient information:** No prior information about the input data was given for Assignment purposes. This may cause the not-full understand of the background and detailed information of the dataset and its purposes.
2. **Computing resource:** Computing resource are limited due to the condition of experiment environment and it might affect on the performance and design of model.
3. **Restrained method:** Though this study is purposely limited in the choice of deep learning package in python to focus on its goal, it still in some degree constrains the potential of the model and the way of designing a better structured model and codes.
4. **Knowledge and capability:** It should be acknowledge that this study is completed within the boundary of our share knowledge and skills, though that enrich while working on thi study, the final outcomes still have room for more optimization, and it is desired in the future that a better version of best model can be discovered.

5. **Assumptions&results:** While experimenting, we make a lot of assumptions about the performance for each change, with some of the assumption accurate and some out of expectation. Our group view this as a good sign as it shows us the process of understanding the principle and effect of different components in a deep learning model.

5 Conclusion

In this project, we created our own deep learning modules to predict a class label for dataset with 50,000 training samples and 10 classes. Different values of parameters were used to build models and evaluated model performance to find the design of the best model, that is, the model with the highest accuracy and lowest loss. The results uncover that using 120 neurons in each layer, ReLU activation function, no momentum, no weight decay, and mini-batch size of 100 with normalisation applied in the pre-processing step best predicts the class label.

6 Appendix

6.1 Code link

Below is the Google Drive link that stores our python codes used in this analysis.

`https://colab.research.google.com/drive/1nyIc1V6_EGNGMF---MMkWaj1-LSIhHPf?usp=sharing`

6.2 Specifications

The specifications for running the codes and perform experiments are shown below.

6.2.1 How to run the code

1. Open the above Google Colab link to access our code .
2. Open our .ipynb Assignment 1 code file on Google Colab, and change directory to your own path of the dataset (test_data.npy, train_data.npy, test_label, train_label.npy) to load the data (note that the data is stored in the below link).
3. Execute codes by clicking ”Run All” in the ”Runtime” drop-down menu on Google Colab.

Please be noted in our implementation, our data is correctly load as required. `https://drive.google.com/drive/folders/1W9O6Vx109_lVZYAFUZiYiUsVcjfGyAJS?usp=drive_link`

6.2.2 Software

1. **Coding Language:** Python
2. **Web-interface:** Google Colab

6.2.3 Hardware

1. **Operating System:** Windows 11
2. **Central Processing Unit(CPU)**: AMD Ryzen 7 5800H with Radeon Graphics 3.20 GHz
3. **Installed Random Access Memory(RAM):** 16.0 GB (15.4 GB usable)

References

[1] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[2] Aniruddha Bhandari. Feature scaling: Engineering, normalisation, and standardization. *Analytics Vidhya*, 2024.

[3] Muhammad Uzair and Noreen Jamil. Effects of hidden layers on the efficiency of neural networks. In *2020 IEEE 23rd international multitopic conference (INMIC)*, pages 1–6. IEEE, 2020.

[4] Christian Janiesch, Patrick Zschech, and Kai Heinrich. Machine learning and deep learning. *Electronic Markets*, 31(3):685–695, 2021.

[5] Dimitris Stathakis. How many hidden layers and nodes? *International Journal of Remote Sensing*, 30(8):2133–2147, 2009.

[6] Alan J Thomas, Miltos Petridis, Simon D Walters, Saeed Malekshahi Gheytassi, and Robert E Morgan. Two hidden layers are usually better than one. In *Engineering Applications of Neural Networks: 18th International Conference, EANN 2017, Athens, Greece, August 25–27, 2017, Proceedings*, pages 279–290. Springer, 2017.

[7] Zuowei Shen, Haizhao Yang, and Shijun Zhang. Neural network approximation: Three hidden layers are enough. *Neural Networks*, 141:160–173, 2021.

[8] Mei Liu, Liangming Chen, Xiaohao Du, Long Jin, and Mingsheng Shang. Activated gradients for deep neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 34(4):2156–2168, 2021.

[9] Hong Hui Tan and King Hann Lim. Vanishing gradient mitigation with deep learning neural network optimization. In *2019 7th international conference on smart computing & communications (ICSCC)*, pages 1–4. IEEE, 2019.

[10] Fabian Schilling. The effect of batch normalization on deep convolutional neural networks, 2016.

[11] Ruo-Yu Sun. Optimization for deep learning: An overview. *Journal of the Operations Research Society of China*, 8(2):249–294, 2020.

[12] Kazuyuki Hara, Daisuke Saito, and Hayaru Shouno. Analysis of function of rectified linear unit used in deep learning. In *2015 international joint conference on neural networks (IJCNN)*, pages 1–8. IEEE, 2015.

[13] Lu Lu, Yeonjong Shin, Yanhui Su, and George Em Karniadakis. Dying relu and initialization: Theory and numerical examples. *arXiv preprint arXiv:1903.06733*, 2019.

[14] Twan Van Laarhoven. L2 regularization versus batch and weight normalization. *arXiv preprint arXiv:1706.05350*, 2017.

[15] Anders Krogh and John Hertz. A simple weight decay can improve generalization. *Advances in neural information processing systems*, 4, 1991.

[16] Maksym Andriushchenko, Francesco D'Angelo, Aditya Varre, and Nicolas Flammarion. Why do we need weight decay in modern deep learning? *arXiv preprint arXiv:2310.04415*, 2023.

[17] Saad Hikmat Haji and Adnan Mohsin Abdulazeez. Comparison of optimization techniques based on gradient descent algorithm: A review. *PalArch's Journal of Archaeology of Egypt/Egyptology*, 18(4):2715–2743, 2021.

[18] Chang Xu. Optimization for training deep models. Course materials of COMP5329, USYD, 2024.

[19] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[20] Chang Xu. Regularizations for deep models. Course materials of COMP5329, USYD, 2024.

[21] Chang Xu. Multilayer neural network. Course materials of COMP5329, USYD, 2024.

[22] Anqi Mao, Mehryar Mohri, and Yutao Zhong. Cross-entropy loss functions: Theoretical analysis and applications. In *International Conference on Machine Learning*, pages 23803–23828. PMLR, 2023.

[23] Solomon Kullback. Kullback-leibler divergence, 1951.

[24] Tracy Chang. Implementing batch normalisation in python. *Towards Data Science*, 2020.

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

[26] Tiancheng Deng. Effect of the number of hidden layer neurons on the accuracy of the back propagation neural network. *Highlights in Science, Engineering and Technology*, 74, 2023.

[27] Sandeep Kumar. Comparison of sigmoid, tanh and relu activation functions. *AITUDE*, 2020.

[28] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Advances in Neural Information Processing Systems*, 2012.