COMP36212 Assignment EX3: Gradient-Based Optimisation

Problem Definition

This assessment explores the use of optimisation to train an artificial neural network to perform image classification. The artificial neural network (ANN) in question is a multi-layer perceptron, and will be used to classify images of hand-written digits (numbers 0-9) from the MNIST dataset. The goal of the assignment is to construct an optimisation algorithm to train the ANN to maximise classification accuracy.

This introduction provides an overview of how the neural network computes a prediction from a sample input from the dataset (Sec. 1); along with an overview of how the network prediction is compared with the true label in order to construct an objective function for optimisation, and details of the target optimisation approaches (Sec, 2). The assessment is then detailed in Sec. 4, providing guidelines for how to approach the optimisation task and construct the report for submission.

Example code is provided to aid completion of the assessment (discussed in Sec. 3), which provides access to the training/testing datasets (summarised in Sec. 1.1), contains the core neural network functionality (summarised in Sec. 1.2), and implements a skeleton optimisation framework. The goal of the assignment is to build on this example, and explore how different optimisation approaches impact performance, both in terms of accuracy on the classification task, and the computational cost of training.

1 Artificial Neural Network

An artificial neural network (ANN) is made up of layers of neurons interacting through weighted connections known as synapses. The type of network explored here is a multi-layer perceptron, also known as a fully-connected network. There are five layers in total in the neural network: the input layer containing 784 neurons; three hidden layers containing 300, 100 and 100 neurons respectively; and an output layer containing 10 neurons. Each neuron in a layer is connected to all neurons in the previous layer through individually-weighted synapses, where each weight can be denoted w_{ij}, where i is the index of the neuron in one layer, and j the index of the neuron in the next layer. The goal of the optimisation problem is to find the individual values of w_{ij} which minimise the classification prediction error of the system.

1.1 ANN Input – MNIST Dataset

The MNIST dataset containing images of handwritten digits is used to train and test the ANN. The dataset contains hand-written digits from 0-9, with sample images shown in Fig. 1. The grey-scale images are defined as 28×28 pixels, with each pixel describing an intensity value in the range 0-255 [1]. The dataset is labelled enabling supervised learning, and split into 60000 sample images for training, and 10000 sample images for testing.

To input an individual image sample to the network, the square 28×28 pixel grid is flattened into an array of length 784, by sequentially concatenating the row-wise pixel descriptions. Each pixel intensity is

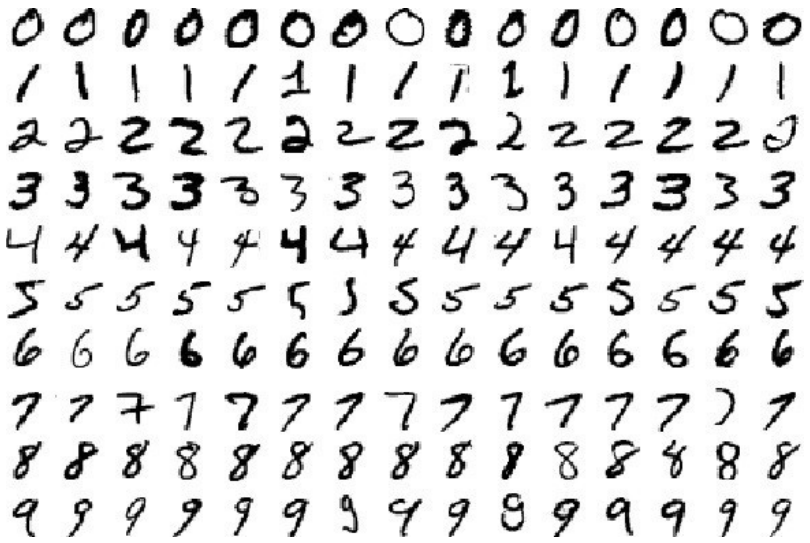

Figure 1: Example hand-written digits from the MNIST dataset [1]

then divided by 255 to normalise inputs to the range $0 \leq x_i \leq 1$, and fed into the network via the input layer x (designated by the blue neurons on the left-hand side of Fig. 2).

1.2 ANN Processing – Neuron Activation

Network input is processed by the ANN layer-by-layer. Each neuron in the network has its own independent activation value: for the input layer this is specified by the presented image sample (see Sec. 1.1), while for the hidden and output layers this is computed according to the activation function. In this work, all hidden neurons use the ReLU (Rectified Linear Unit) activation function, while the output units use the softmax activation function. For each neuron, its activation is designated as h_j^n, where n gives the layer index, and j the neuron index within the layer – e.g. h_{45}^2 is the 45th neuron in the second hidden layer.

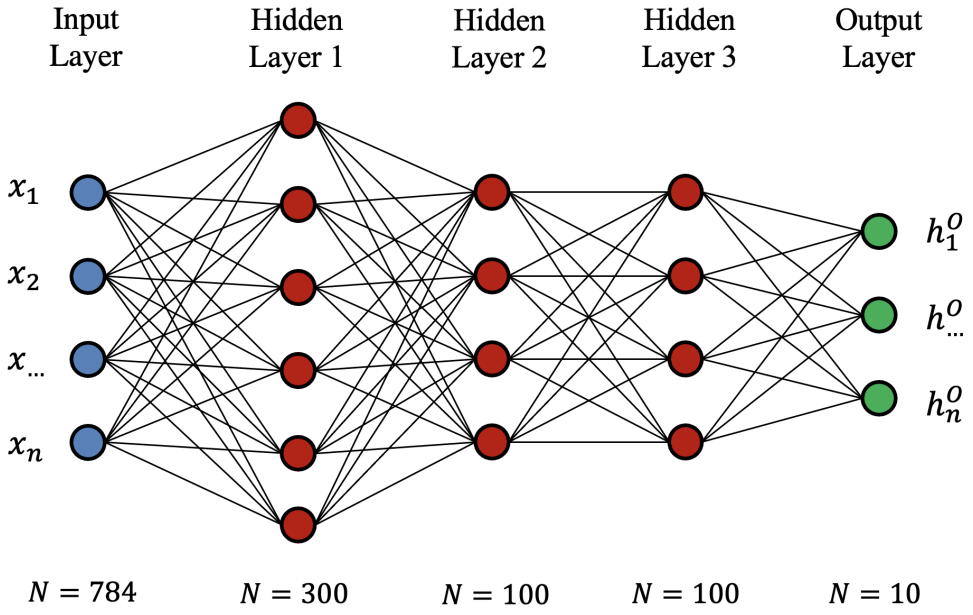

Figure 2: Multi-layer artificial neural network (ANN) for classifying images of hand-written digits 0-9

Hidden neuron activation values are calculated in two steps, first by evaluating the weighted sum of the inputs from the previous layer to compute z_j^n; then passing this weighted sum through the ReLU (Rectified Linear Unit) activation function to compute h_j^n, as described in Eqn. 1.

$$h_j^n(h^{n-1}) = \begin{cases} z_j^n & \text{if } z_j^n > 0 \\ 0 & \text{otherwise} \end{cases} \qquad \text{where} \quad z_j^n = \sum_{i=0}^{N} w_{ij} h_i^{n-1} \tag{1}$$

Equation 1, describes the activation of the three hidden layers within the network (h^1 to h^3), indicated in red in Fig. 2. For the first hidden layer, h^{n-1} is replaced with input x in the calculation of z^1. The output layer (shown in green in Fig. 2) performs the same weighted sum (using h^3 as input), however the softmax activation function is used in place of the ReLU activation function, as discussed in Sec. 1.3.

1.3 ANN Output – Classification Predictions

The goal of the ANN is to classify the individual digit presented in an input image, meaning the network must predict a number from 0-9. This is achieved by having 10 output neurons, with the network output interpreted as the activation of these output neurons. To interpret the prediction, the output layer of the network calculates neuron activations as discussed previously, but additionally performs the *softmax* operation, to turn the logits (individual output activations) into a discrete probability distribution.

$$S_j = \frac{e^{h_j^O}}{\sum_{k=0}^{N} e^{h_k^O}} \qquad \text{where } P_j = S_j \tag{2}$$

The softmax operation is shown in Eqn. 2, which transforms the output vector h^O into a vector S, which has all values normalised between 0 and 1, with the sum of all values equal to 1 (as required by a probability distribution). Given an input x, the layer-by-layer activations can be calculated, and the network prediction evaluated as S - e.g. the probability that the network is predicting the digit zero is given by $P_{j=0}$, while the overall prediction from the network is given by: $\max(P)$.

2 Optimisation

The parameters for optimisation are the ANN connection weights. Neurons are connected in an all-to-all fashion, with each connection having an independent synaptic weight w_{ij}, linking neuron i in one layer to neuron j in the next. These weights form the variables in the optimisation problem, and will be updated to minimise classification error. This results in $28 \times 28 \times 300 = 235200$ weights between the input and first hidden layer, 30000 weights between the first two hidden layers, 10000 weights between the second and third hidden layers, and 1000 weights between the third hidden layer and output – giving a total of 276200 parameters within the optimisation problem.

2.1 Initialisation

The input data, and computational steps taken by the ANN to process it, are well defined. However, in order to make an initial prediction with the ANN, the weight parameters defining the neuron-neuron connections must be initialised. This initialisation step requires careful consideration, as it can bias the optimisation process towards a particular solution, or can result in such poor initial performance that the system can go unstable and prevent training at all.

In this work, weights are initialised using a uniform distribution, with limits set according to Glorot et al [2], as shown in Eqn. 3. Here U denotes a uniform distribution, with m the number of neurons in the preceding layer, and n the number of neurons in the current layer.

$$w_{ij} = U\left(-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}}\right) \tag{3}$$

This initialisation provides similar activity in each layer despite the varying number of neurons in preceding layers. This creates a useful baseline activation level in all neurons, enabling successful training using the approaches discussed below.

2.2 Loss Function

While the output layer provides the predicted classification of the network, an additional step is required to enable optimisation and training of the network. This assessment deals with supervised learning, and updates the ANN based on how the predicted output compares with the true output. The true output is defined within the MNIST dataset, meaning each time an input sample is presented, the ANN prediction can be compared with the correct value. This comparison is evaluated in terms of a loss function L.

To compare the network prediction with the true labelled value from the dataset, the label is transformed into a one-hot encoded vector representing the target probability distribution. This is a vector with the same number of entries as the output layer of the network, with the value 1 in the position of the labelled class, and zeros elsewhere. The difference between this vector and the output from the ANN for the given sample is then calculated according to the cross-entropy loss, as shown in Eqn. 4.

$$L = -\sum_{k=0}^{N} \hat{y}_k \log(P_k) \tag{4}$$

Where N is the number of possible classes (10 in the case of the MNIST dataset), $\hat{y}$ is the correct class label (1 for the correct class, 0 otherwise), and P is the network prediction for the given class (calculated via Eqn. 2).

2.3 Evaluating Network Performance

While the loss function captures the prediction error for an individual sample, it doesn't provide a complete picture of how the ANN will perform overall. To compute a metric capturing overall performance, the testing set of 10000 images is presented to the ANN, and the number of correct predictions is recorded. The ANN accuracy is then calculated according to:

$$\text{Accuracy} = \frac{\text{Total correct predictions}}{\text{Total predictions}} \tag{5}$$

The provided code periodically evaluates the ANN on the testing set in order to monitor system performance. While there are other techniques available to evaluate the performance during training (such as using a small subset of the training set), this approach is used here for simplicity.

2.4 Gradient-Based Optimisation

Training of the ANN will be performed by optimisation. To formulate the training problem as a minimisation task, the loss L must be transformed into an objective function capturing the ANN performance on all samples in the dataset (not a single sample as evaluated in Eqn. 4). The objective function is

therefore constructed from the loss L, evaluated over the entire dataset, as shown in Eqn. 6, where n is the number of samples in the dataset (60000 for the MNIST training set).

$$f(w) = \frac{1}{n}\sum_{i=1}^{n} L_i(x_i, w) \tag{6}$$

$$\nabla f(w) = \frac{1}{n}\sum_{i=1}^{n} \nabla L_i(x_i, w) \tag{7}$$

The true objective function $f(w)$ (Eqn. 6) is calculated as the average loss over the entire dataset (where i, here refers to sample index within the dataset). Training the system involves minimisation of $f(x)$, which can be achieved using optimisation algorithms. The *gradient descent* algorithm requires evaluation of the gradient of the objective function $f(w)$, with respect to each individual optimisation parameter w and is denoted as $\nabla f(w)$. For the true gradient descent algorithm, $\nabla f(w)$ is evaluated based on the entire dataset. The ANN weights can then be updated via the gradient descent algorithm, as shown in Eqn. 8, where η is the learning rate.

$$w = w - \eta \nabla f(w) \tag{8}$$

Optimistion typically requires multiple passes through the training dataset in order to converge on an optimum. It is therefore common to discuss convergence in terms of *epochs*, where one epoch consists of a single pass through the dataset.

2.5 Stochastic Gradient Descent

Evaluating $\nabla L(w)$ over the entire dataset is an expensive operation computationally, and can be prohibitive when the dataset is large. A significant development in training ANN models was the discovery that estimating the gradient from a subset of the dataset can privde a sufficiently accurate representation to facilitate optimisation [3]. When a single sample is chosen at random from the dataset, and used to evaluate ∇L and update the weights, the optimisation algorithm is known as *stochastic gradient descent* (SGD). A common variation of SGD is *batch SGD*, which uses a subset of samples from the dataset to compute $\nabla L(w)$, where m in Eqn. 9 is now the size of the batch, rather than the entire dataset.

$$w = w - \frac{\eta}{m}\sum_{i=1}^{m} \nabla L_i(x_i, w) \tag{9}$$

This means that the weight update is applied only after processing the complete batch, averaging weight updates across the different samples within the batch. This is beneficial, as m in Eqn. 9, is typically significantly smaller than n in Eqn. 8, meaning SGD can begin updating parameters much quicker than basic gradient descent. More importantly, it means the performance of SGD is not defined by the size of the training dataset, meaning the algorithm scales well to extremely large problems.

2.6 Learning Rate Decay

The learning rate η governs the maximum step size which can be taken by an optimisation algorithm, and hence has a significant impact on convergence, making it one of the most important hyper parameters in optimisation problems. However, it can be a challenge to find a learning rate which both provides rapid initial convergence (e.g. where large learning rates would be beneficial), and local refinement close to the optimum (e.g. where smaller learning rates would be more successful). One solution to this challenge is to decay the learning rate over the course of the optimisation to match the problem requirements. A method to provide linear learning rate decay is demonstrated in Eqn. 10.

$$\eta_k = \eta_0(1-\alpha) + \alpha\eta_N \qquad \text{where } \alpha = \frac{k}{N} \tag{10}$$

Here the learning rate η_k is a function of: the initial learning rate η_0, and final learning rate to η_N, where N is the total number of epochs, and k is the current epoch number. The learning rate is updated at the beginning of each epoch, with $\alpha = {}^{k}/_{N}$.

2.7 Momentum

Slow convergence near an optimum is a feature of gradient-descent algorithms, as they can get trapped in local features, such as local minima and regions of pathological curvature. In these regions, the gradients of certain parameters with respect to the loss function can oscillate between different directions, and dominate shallower true convergence gradients, extending the time taken (and sometimes preventing) to reach an optimum. The concept of momentum can be used to accelerate learning, and overcome these challenges, through the use of an exponentially decaying moving average of past gradients – as shown in Eqn. 11.

$$\begin{aligned} v &= \alpha v - \frac{\eta}{m} \sum_{i=1}^{m} \nabla L_i(x_i, w) \\ w &= w + v \end{aligned} \tag{11}$$

This average is implemented through a velocity term v, which accumulates the moving average of past gradients, and a hyper parameter α, which defines the decay rate of contributions from previous gradient estimates. The step size is therefore now determined by the magnitude of the gradient at the current point, but also how aligned this is with previous gradient estimates. For example, the step size will be greatest when successive gradient estimates are consistently in the same direction, while oscillating estimates will cancel.

3 Code

A simple C program is supplied to aid completion of the assignment. This contains code to preprocess the dataset, initialise and calculate the forward pass through the dataset, calculate gradients of the loss with respect to network weight parameters, and provides a base optimisation framework. The MNIST dataset can also be downloaded from Blackboard, and should be unzipped, and placed at the location: `<path to mnist_data dir>`, as indicated below.

The supplied code can be compiled using a general C compiler, e.g. using gcc:

```
gcc -o mnist_optimiser.out main.c mnist_helper.c neural_network.c optimiser.c -lm -O3
```

and then run from the command line:

```
./mnist_optimiser.out <path to mnist_data dir> <learning rate> <batch size> <num epochs>
```

The `-O3` flag is suggested to help optimise compilation for speed, as the code is compute intensive, and requires many iterations to explore the different optimisation approaches. Note that the code is single-threaded, however if your processor allows multiple instances can be run simultaneously to achieve parallelisation.

`main.c`

Provides a means to pass (and check) input arguments to the optimisation framework; calls the initialisation functions for the dataset handler, neural network and optimiser; and runs the optimisation. To simplify the code, global variables are defined to provide access to the core data structures across the different code files.

`mnist_helper.c`

Provides access to the MNIST dataset, and should not require editing to complete the assessment. This file contains functions to load the dataset files, and parse the input samples and labels. The `mnist_helper.h` file contains key dataset dimensions for use by the rest of the model, and provides access to the loaded testing and training datasets through the arrays `training_data` and `testing_data`, which are both indexed by sample. There are corresponding arrays of correct labels: `training_labels` and `testing_labels`, again indexed by sample.

`neural_network.c`

Contains the definition of the neurons and connections making up the neural network, together with the gradient and Jacobian datastructures required to compute derivatives for the optimisation algorithm.

Arrays of neurons are defined using the structs: `n_ReLU` for hidden layer neurons; and `n_Out` for the output layer (`n_L0`). For the hidden layer neurons, array `n_Lx` denotes the neurons of layer x (where x can take: 1, 2, 3, representing the ANN hidden layers). The matrices of weights connecting the network layers consist of arrays of type `weight_struct_t`, defined in `neural_network.h`. A base type is defined to enable completion of Part I, however it may be useful to define a new type with additional parameters for Parts II & III. The base type contains parameters for the connection weight (`w`), and gradient (`dw`). Arrays of connections are defined between layers, e.g. with `W_LI_L1` defining the connections between the input layer and layer 1.

In addition to the data structures, functions are provided to evaluate the forward pass through the network, the output probability distribution and the resulting loss. Also provided are functions to evaluate the backward pass through the network, analytically calculating gradients of the loss function with respect to the weight parameters: `dL_dW_L3_L0`, `dL_dW_L2_L3`, `dL_dW_L1_L2`, and `dL_dW_LI_L1`. These gradients are then stored into the connections, summing the individual gradients into the `dw` term in `weight_struct_t`. This facilitates implementation of batch optimisation algorithms, and simplifies accessing of the gradient information, as it shares the same index system as connection weights (i.e `w` in `weight_struct_t`). A function is also provided to evaluate the performance of the neural network when applied to the testing set: `evaluate_testing_accuracy()`. This neural network code (both forward and backward passes) should not need to be changed in order to complete the assignment.

`optimiser.c`

Contains the optimisation framework code, managing the number of iterations, epochs and batch size etc. Within this framework there are calls to the functions in `neural_network.c` to evaluate the forward pass through the network, calculate the loss, and evaluate the gradients. Periodically calls are also made to `evaluate_testing_accuracy()` to enable monitoring of overall performance. The provided `optimiser.c` includes a placeholder function `update_parameters()` which can be used to implement the different optimisation algorithms required in Parts I–III of the assessment.

4 Assessment

ANN architectures similar to the one utilised in this work have previously demonstrated good performance on the MNIST digit recognition task, achieving over 97% accuracy on the testing set. While maximising classification accuracy is a fundamental goal of the given problem, the computational challenge of locating the optimum must also be considered. Identifying a suitable optimisation strategy is therefore an important factor in maximising system accuracy, and minimising the computational cost of locating the optimum. The assessment therefore explores and analyses both aspects of optimisation, with the assessment submission taking the form of a short report addressing the following three parts:

Part I - Stochastic Gradient Descent (10 marks)

The supplied code provides a base optimisation framework to train a multi-layer perceptron to recognise images of hand-written written digits from the MNIST dataset. A common method to train this type of system is stochastic gradient descent (SGD) – see Sec. 2.5. The goal of Part I, is to implement and test the performance of SGD as a training algorithm for the given task.

1. Complete the skeleton optimisation framework defined in `optimiser.c`, by implementing batch SGD as described in Eqn. 9. This will require completion of the `update_parameters()` function in `optimiser.c`, defining the update rule for the weight matrices: `W_LI_L1`, `W_L1_L2`, `W_L2_L3` and `W_L3_L0`. Include a concise description of the implemented algorithm in the report.

2. A key step in gradient-based optimisation is calculation of the gradients. While analytical calculation of gradients has been implemented in the supplied code, it is good practice to validate the calculated gradients to ensure correct performance. Use the numerical differentiation techniques explored earlier in the course to verify the analytical gradient calculations. Discuss the method used and results in the report. How does numerical calculation of gradients compare to the analytical method in terms of accuracy and compute time?

3. Run the SGD optimiser and observe how the model is trained, using hyper parameters: batch size $m = 10$, and learning rate $\eta = 0.1$. Plot convergence over 10 epochs, and discuss in the report text how the implementation is verified, ensuring that the parameter updates are performed according to the theoretical description of SGD, and that the optimiser improves ANN prediction accuracy.

4. Discuss the performance of SGD on the given problem using the given hyper parameters - has the algorithm found an optimum? (Justify your answer.)

Part II - Improving Convergence (10 marks)

Convergence of SGD can be slow around an optimum. This section will explore two techniques to enable rapid location of the optimum, and the improving the process of fine-tuning the network when close to the optimum: learning rate decay, and momentum.

1. Explore how batch size (m) and learning rate (η) impact SGD performance. For example, investigate the effect of changing m and η on peak performance, speed of convergence, and stability. A useful set of configurations to explore optimisation for is: batch sizes $m = 1,\ 10,\ 100$; and learning rates $\eta = 0.1,\ 0.01,\ 0.001$. Run each optimisation for 10 epochs (or longer if there is a particular characteristic this will showcase), plotting convergence of testing accuracy as a function optimisation iterations (or epochs) for the different configurations. Analyse and discuss the results in the report text.

2. Implement learning rate decay (e.g. the mechanism described in Sec. 2.6), and explore how this can be used to combine the best attributes of the SGD configurations explored above. Run a range of experiments to explore the effect of method hyper parameters on convergence and overall accuracy. Plot the outcome in the report, and analyse/discuss the effectiveness of learning rate decay as a technique to improve convergence of SGD.

3. Implement a momentum based weight update, e.g. following the approach discussed in Sec. 2.7. Additional parameters can be added to `weight_struct_t` to enable similar indexing to values such as the weights. However, note that it is recommended to initialise to zero any new variables added to `weight_struct_t` in the function `initialise_weight_matrices()` in `neural_network.c` to avoid unpredictable behaviour. Plot the outcome in the report, and analyse/discuss the effectiveness of momentum as a technique to improve convergence of SGD.

4. Explore the combination of learning rate decay and momentum based weight updates, and discuss the optimum hyper-parameters for the given problem. Plot the resulting convergence behaviour, and compare with the results above, providing analysis and discussion in the report text.

Part III - Adaptive Learning (10 marks)

As the learning rate plays such a key role in optimisation performance, identification of the optimal learning rate is the subject of intense research, generating a great range of algorithms capable of tailoring the learning rate during optimisation. As individual parameters can be more or less sensitive to the learning rate during optimisation, based on their current gradients, researchers have looked to apply the concept of parameter-specific learning rates to improve performance.

The goal of this section is to research and implement an optimisation algorithm from the literature which utilises adaptive learning rates, such as: AdaGrad [4], RMSProp [5], Adam [6], or an alternative algorithm of your choosing.

1. Describe in the report the proposed approach mathematically, and discuss theoretically how it is expected to impact performance on the given task.

2. Run experiments exploring the implemented optimisation algorithm, and evaluate performance relative to the algorithms investigated in Parts I & II. Provide results and figures to support your analysis/discussion in the report text.

3. Evaluate the optimal hyper-parameters for the algorithm and draw conclusions on the best optimisation approach for the given task (comparing all techniques from Parts I–III).

Report (5 marks)

You should submit a PDF report demonstrating your solutions to the tasks in Parts I–III. Effort should be spent proportional to the marks awarded for each task. An additional 5 marks will be awarded based on the quality of the report, giving an overall total of 35 marks for the assignment. The report should contain concise technical answers expressing your findings and reasoning on the given tasks, and include high-quality figures plotting optimisation results as instructed. The 5 additional marks will be awarded according to the criteria in Tab. 1.

Please submit your code containing implementation of the optimisation algorithms as part of your submission (it is fine to zip up your code together with the report PDF and submit this). Note that **no marks will be awarded for the code** with this assignment, but it must be submitted to validate how you obtained the results. Therefore **the report should contain all information pertaining to solution of the tasks**, and should be **no more than 8 A4 pages in length** (minimum 10pt font size, and margins ≥ 2 cm), however references in a bibliography are not included in this limit.

References

[1] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[2] 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.

[3] Léon Bottou and Olivier Bousquet. The tradeoffs of large scale learning. In J.C. Platt, D. Koller, Y. Singer, and S. Roweis, editors, *Advances in Neural Information Processing Systems 20 (NIPS 2007)*, pages 161–168. NIPS Foundation (http://books.nips.cc), 2008.

[4] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *J. Mach. Learn. Res.*, 12:2121–2159, 2011.

[5] Ian J. Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, Cambridge, MA, USA, 2016.

[6] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *arXiv*, 2011.

Section	Task	$0-25\,\%$	$26-50\,\%$	$51-75\,\%$	$76-100\,\%$
Part I (10 marks)	Stochastic Gradient Descent	Demonstrate knowledge of the SGD algorithm and how the underlying theory relates to the target problem domain	Demonstrate how the theoretical algorithm can be converted into a form suitable for implementation in software	Apply correctly the SGD optimisation algorithm to train an ANN, providing in-depth discussion, analysis and validation of results/performance	Exceptional implementation, analysis, discussion and validation, going beyond the scope of the assignment brief, and drawing on the scientific literature.
Part II (10 marks)	Improving Convergence	Demonstrate how to analyse the effect of hyper parameters on algorithm convergence, linking the theoretical principles of an optimisation algorithm to the target problem domain	Demonstrate how additional mechanisms can be introduced to improve convergence, exploring how a theoretical method can be converted into a form suitable for implementation in software	Implement correctly techniques to improve convergence and apply them to the target problem domain, providing in-depth discussion and analysis of results/performance	Exceptional implementation, analysis and discussion, going beyond the scope of the assignment brief, and drawing on the scientific literature.
Part III (10 marks)	Adaptive Learning	Select an algorithm to implement enabling adaptive learning rates, motivating the selection in the context of the target problem domain	Demonstrate how theoretical algorithm can be converted into form suitable for implementation in software, and apply it to train the target ANN	Explore tuning and performance of the implemented algorithm. Based on data from experiments, critique the overall analysis in the context of the given task, discussing strengths/limitations, and how the approach could generalise to other problems	Exceptional implementation, analysis and discussion, going beyond the scope of the assignment brief, and drawing on the scientific literature.
Report (5 marks)	Compose research report summarising the methods and implementation, analysing results and discussing performance	Introduce methods and techniques, and discuss results, providing supporting information to validate analyses, including summary conclusions where appropriate	Correct use of mathematical notation and numbering of equations, and high quality presentation of data in figures and tables	Provide in-depth analysis of results and describe techniques such that a reader of the report can interpret/reproduce the methods	Exceptional analysis, discussion and conclusions, backed up by numerical findings and theory, ensuring work is appropriately referenced

Table 1: Assessment rubric for COMP36212-EX3 (to achieve a given level, all attributes/skills to the left must also have been demonstrated).