Project 2a: Neural Networks[1]

1 Introduction

The idea that computers may be able to think or exhibit intelligence dates back to the dawn of the computing era. Alan Turing[2] argued in his seminal 1950 paper *Computing Machinery and Intelligence* that digital computers should be capable of thinking, and introduced the famous 'Turing test' to define what was meant by 'thinking' in this context: the computer should be able to emulate a human mind. He also suggested that a computer exhibiting human intelligence could not be programmed in the usual way (where the programmer has 'a clear mental picture of the state of the machine at each moment in the computation', as he put it) but would instead learn like a human, drawing inferences from examples. Turing had previously identified that learning of this sort could occur in what he called an *unorganised machine*: a computational model of a brain consisting of many simple logical units connected together in a network to form a large and complex system.

Although a computer emulation of a complete human mind is still some way off, Turing's idea that a network could be taught and learn was prescient. Models of this sort are now known as Artificial Neural Networks (ANNs), and the process by which they learn from examples is known as supervised learning, part of the large field of machine learning. In the last decade or so, ANNs have developed to become the best algorithms available for a range of 'hard' problems, including image recognition and classification, machine translation, text-to-speech, and playing of complex games such as Go.

In this project we will develop a simple neural network and train it to classify data into one of two classes, a so-called *binary classification* problem. (This problem is discussed further by Higham and Higham (2019) whose notation we use in this project.) Suppose we have a set of points in $\mathbb{R}^2$, as shown in Fig. 1, acquired from some real-world measurements, with each point being of one of two classes, either a blue cross or a red circle:

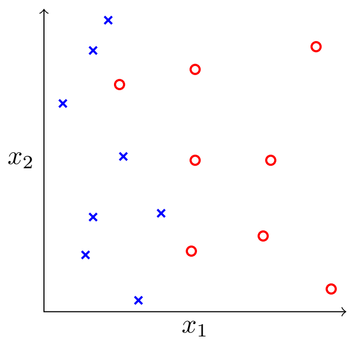

Figure 1: A set of data points in $\mathbf{R}^2$, each of which is of one of two types.

This type of data could arise in many applications, for example the success or failure of an industrial chemical process as a function of ambient temperature and humidity, or

[1]Project originally developed by Chris Johnson; modified by Matthias Heil.

[2]At the time of writing that paper, Turing was a Reader in this department, at what was then called the Victoria University of Manchester

whether a customer has missed a loan payment as a function of their income and credit score.

Given a new point $\mathbf{x} \in \mathbb{R}^2$, we'd like to be able to predict from the existing data which of the two classes this new point is likely to be in. Since our example data shows a clear pattern (red circles tend to lie at larger values of x_1 than blue crosses), we could sketch a curve by eye (as in Fig. 2) that divides the plane into two, with the red circles on one side and the blue crosses on the other.

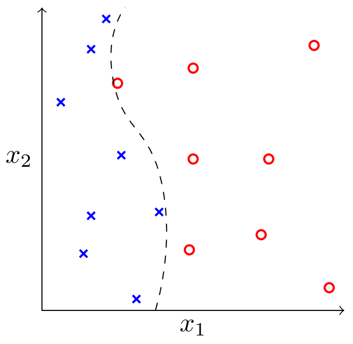

Figure 2: The plane is divided by a curve, with points on each side classified differently. Given a new point on the plane, we can now estimate which class it is in by seeing which side of the line it lies.

We could then classify any new point $\mathbf{x} \in \mathbb{R}^2$ by seeing on which side of the curve it lies. Our neural network will perform this classification for us by (1) learning from existing data where the boundary between the two classes of data point lies, and (2) allowing us to evaluate the likely class of a new data point by using the boundary found in step (1). Specifically, the network defines a function $F(\mathbf{x})$, where the curve dividing our two types of point is the zero contour $F(\mathbf{x}) = 0$. Regions where the network outputs a positive value $F(\mathbf{x}) > 0$ correspond to one class, and regions where it is negative to the other class.

Although this two-dimensional problem is relatively simple, the neural network developed extends naturally to much more difficult classification problems in higher dimensions (where $\mathbf{x}$ has hundreds or thousands of components), with more than two classes of points, and where the boundaries between different regions are not as clear as in our example above. One example is character recognition, as used by banks and postal services to automatically read hand-written cheques and postcodes. In this case, a segmentation algorithm is applied to split the scanned text into lots of small images each containing a single digit. Suppose each such small image has $25 \times 25 = 625$ greyscale pixels; the data in it can be represented by a vector $\mathbf{x} \in \mathbb{R}^{625}$. A neural network of exactly the sort we develop can be used to classify a handwritten character encoded in the vector $\mathbf{x}$ into one of 36 classes (one for each of the characters 0–9 and A–Z) with very good accuracy.

2 Neural Networks: Theory

2.1 Overall setup

An artificial neural network consists of a set of *neurons*. A neuron is a simple scalar functional unit with a single input and a single output, modelled by an nonlinear activation function

$\sigma : \mathbb{R} \to \mathbb{R}$. In the *feed-forward* networks we will study, these neurons are arranged in a series of layers, with the outputs of neurons in one layer connected to the inputs of neurons in the next, as shown in Fig. 3. The networks in this project will also be *fully connected*, meaning that nodes in two adjacent layers and the connections between them form a complete bipartite graph (every neuron in one layer is connected to every neuron in the next).

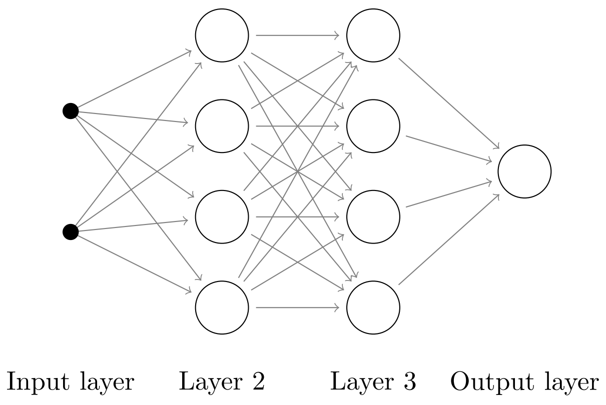

Figure 3: Fully-connected network topology with two input neurons, two hidden layers of four neurons each, and one output neuron.

The inputs to the neural network are specified at the first layer; in our classification problem, this consists of the two components of the input vector $\mathbf{x} = (x_1, x_2)$. The output of the network is the output of the final layer of neurons; in our example this is just a single neuron, outputting a real number that we hope is close to either $+1$ or -1, depending on the class of the point. In between are one or more *hidden layers*. The network as a whole defines a nonlinear function, in this case $\mathbb{R}^2 \to \mathbb{R}$ (two inputs, one output).

As noted above, there are two stages to using a neural network.

1. The network is *trained* by specifying both the input to the network and the desired (target) output. Through an iterative procedure ("learning"), parameters relating to the connections between neurons (known as the weights and biases, defined below) are modified so that the network reproduces the target output for a wide range of input data. The overall structure of the network (the number of layers, and the number of neurons in each layer) is not changed in the training.

2. The weights and biases are then fixed, and the output of the network is evaluated for arbitrary new input data.

We will consider initially the second of these steps, assuming that the network has been pre-trained and that the weights and biases are known.

2.2 Evaluating the neural net output

Since each neuron is a simple scalar function ($\mathbb{R} \to \mathbb{R}$), the multiple inputs into a neuron from neurons in the previous layer must be combined into a single scalar input. To describe how this is done we first establish some notation. Let L represent the total number of layers in the network and n_l represent the number of neurons at layer l, for $l = 1, 2, \ldots, L$. The input to the jth neuron at layer l is denoted by $z_j^{[l]}$ and the output from the jth neuron at layer l by $a_j^{[l]}$. The statement that each neuron is modelled by an activation function σ can then be written algebraically as

$$a_j^{[l]} = \sigma_l\left(z_j^{[l]}\right), \qquad \text{for } l = 2, 3, \ldots, L, \quad j = 1, \ldots, n_l, \tag{1}$$

where the subscript on the activation function implies that different layers may use different activation functions (but all the neurons in a layer use the same one). The statement that the inputs to the neural network are specified at the first layer can be written as

$$a_j^{[1]} = x_j, \quad j = 1, \ldots, n_1. \tag{2}$$

Note that the input to the network x_j determines the outputs $a_j^{[1]}$ of the neurons in the first layer rather than their inputs, so (1) is not evaluated for the first layer $l = 1$. The output of the network is the output of the final layer of neurons, $a_j^{[L]}$. Consequently, the network has n_1 (scalar) inputs and n_L (scalar) outputs. The notation can be simplified by defining the neuron inputs and outputs as vectors $\mathbf{z}^{[l]}, \mathbf{a}^{[l]} \in \mathbb{R}^{n_l}$, with components $z_j^{[l]}$ and $a_j^{[l]}$ respectively. Then, defining the activation function to act componentwise, (1) becomes

$$\mathbf{a}^{[l]} = \sigma_l\left(\mathbf{z}^{[l]}\right), \qquad \text{for } l = 2, 3, \ldots, L. \tag{3}$$

The activation function is typically a nonlinear monotonically increasing function, but may have one of a number of different functional forms. One common choice is the tanh function,

$$\sigma(z) = \tanh(z). \tag{4}$$

While (3) and (4) together define the behaviour of the individual neurons, the more significant part of a neural network is the set of connections between neurons, which determine how the input $z_j^{[l]}$ (of a neuron j in layer l) is determined from the outputs of the neurons in the previous layer. Specifically, a neuron input $z_j^{[l]}$ is a biased linear combination of the outputs of the neurons in layer $l - 1$,

$$z_j^{[l]} = \sum_{k=1}^{n_{l-1}} \left(w_{jk}^{[l]} a_k^{[l-1]}\right) + b_j^{[l]}, \qquad \text{for } l = 2, 3, \ldots, L, \quad j = 1, \ldots, n_l, \tag{5}$$

where $w_{jk}^{[l]}$ are the *weights* and $b_j^{[l]}$ are the *biases* at layer l. These weights and biases are adjusted while the network is learning from the training data, but thereafter remain fixed. Defining $\mathbf{b}^{[l]} \in \mathbb{R}^{n_l}$ as the vector of biases and $\mathbf{W}^{[l]} \in \mathbb{R}^{n_l} \times \mathbb{R}^{n_{l-1}}$ as the matrix of weights at layer l, with components $b_j^{[l]}$ and $w_{jk}^{[l]}$ respectively, we can write (5) as

$$\mathbf{z}^{[l]} = \mathbf{W}^{[l]}\mathbf{a}^{[l-1]} + \mathbf{b}^{[l]}, \qquad \text{for } l = 2, 3, \ldots, L. \tag{6}$$

Combining this with (3) and (2) we find

$$\mathbf{a}^{[1]} = \mathbf{x}, \tag{7}$$

$$\mathbf{a}^{[l]} = \sigma_l\left(\mathbf{z}^{[l]}\right) = \sigma_l\left(\mathbf{W}^{[l]}\mathbf{a}^{[l-1]} + \mathbf{b}^{[l]}\right) \qquad \text{for} \quad l = 2, 3, \ldots, L. \tag{8}$$

Equations (7) and (8) together describe the *feed-forward* algorithm for evaluating the output of a neural network given an input vector $\mathbf{x}$ (and known weights $\mathbf{W}^{[l]}$ and biases $\mathbf{b}^{[l]}$). First (7) is used to define $\mathbf{a}^{[1]}$, then (8) is used for $l = 2, 3, \dots, L$ to obtain the neuron outputs $\mathbf{a}^{[2]}, \mathbf{a}^{[3]}, \dots, \mathbf{a}^{[L]}$ in turn. The last of these, $\mathbf{a}^{[L]}$, is the output of the neural network.

2.3 Training the neural network

2.3.1 Gradient descent

The feed-forward algorithm outlined in the previous section allows the output of the neural network to be evaluated from its inputs, provided that the weights and biases at each layer of the network are known. These parameters are determined by training the neural network against measurements or other data for which both the input and target output of the network are known.

This training data consists of a set of N inputs to the network, $\mathbf{x}^{\{i\}}$, with N corresponding target outputs, denoted $\mathbf{y}^{\{i\}}$ $(i = 1, \dots, N)$. We can formalise the idea of how closely our network produces the target outputs $\mathbf{y}^{\{i\}}$ (when given the corresponding inputs $\mathbf{x}^{\{i\}}$) by defining a total cost function,

$$C = \frac{1}{N} \sum_{i=1}^{N} \frac{1}{2} \|\mathbf{y}^{\{i\}} - \mathbf{a}^{[L]}(\mathbf{x}^{\{i\}})\|_2^2, \tag{9}$$

where ‘total cost’ refers to this cost being the sum of costs over all the N training data, and the squared L_2 norm $\|\cdot\|_2^2$ of a vector is the sum of its components squared,

$$\|\mathbf{x}\|_2^2 = x_1^2 + x_2^2 + \cdots + x_n^2. \tag{10}$$

In the cost function (9), for each training datum i the norm is taken of the difference between the target network output $\mathbf{y}^{\{i\}}$ and the actual output of the neural network $\mathbf{a}^{[L]}(\mathbf{x}^{\{i\}})$, evaluated from the corresponding network input $\mathbf{x}^{\{i\}}$ by using the feed-forward algorithm. If the network reproduces exactly the target output for every piece of input data, the cost function is equal to its minimum possible value, zero.

For a given set of training data $(\mathbf{x}^{\{i\}}, \mathbf{y}^{\{i\}})$ $(i = 1, ..., N)$, the cost C is a function of the parameters of the neural network, namely all the weights and biases for all of the layers. We could write this dependence explicitly by using (8) recursively to express $\mathbf{a}^{[L]}(\mathbf{x}^{\{i\}})$ in terms of $\mathbf{a}^{[1]} = \mathbf{x}^{\{i\}}$ and the weights and biases of layers 2 to L,

$$\begin{aligned} C(\mathbf{W}^{[2]}, \dots, \mathbf{W}^{[L]}, \mathbf{b}^{[2]}, \dots, \mathbf{b}^{[L]}) &= \frac{1}{N} \sum_{i=1}^{N} \frac{1}{2} \|\mathbf{y}^{\{i\}} - \mathbf{a}^{[L]}\|_2^2 \\ &= \frac{1}{N} \sum_{i=1}^{N} \frac{1}{2} \|\mathbf{y}^{\{i\}} - \sigma_L \left(\mathbf{W}^{[L]} \sigma_{L-1} \left(\mathbf{W}^{[L-1]} \sigma_{L-2}(\dots) + \mathbf{b}^{[L-1]} \right) + \mathbf{b}^{[L]} \right) \|_2^2. \end{aligned} \tag{11}$$

The network sketched above with $n_l = (2, 4, 4, 1)$ has in total 9 biases (one for each hidden and output neuron) and 28 weights (one for each connection between a pair of neurons), making the cost C dependent on 37 parameters in total. For ease of notation we denote by $\mathbf{p} \in \mathbb{R}^M$ the vector containing the weights and biases at each layer that parameterise the network, where M is the number of such parameters (in this case $M = 37$). We can then write $C = C(\mathbf{p})$ without needing to refer to the weights and biases explicitly.

The process of training the network is equivalent to minimising the difference between the desired (target) and actual neural network outputs for the training data, or in other words, choosing $\mathbf{p}$ (and thus the weights and biases at each layer) so as to minimise the cost $C(\mathbf{p})$, which is a nonlinear function of $\mathbf{p}$.

We will solve this problem using an iterative method based on the classical technique of steepest descent. Suppose we have a current vector of parameters $\mathbf{p}$ and initial cost $C(\mathbf{p})$, and wish to generate a new vector of parameters $\mathbf{p} + \delta\mathbf{p}$ such that the new cost $C(\mathbf{p} + \delta\mathbf{p})$ is minimised. What value should we choose for $\delta\mathbf{p}$? For small $\delta\mathbf{p}$ we can use a Taylor series to expand the new cost

$$C(\mathbf{p} + \delta\mathbf{p}) \approx C(\mathbf{p}) + \sum_{r=1}^{M} \frac{\partial C(\mathbf{p})}{\partial p_r} \delta p_r, \tag{12}$$

where p_r is the rth component of the parameter vector $\mathbf{p}$, and terms of order $(\delta\mathbf{p})^2$ and higher have been neglected. Equivalently, we can write this in vector form,

$$C(\mathbf{p} + \delta\mathbf{p}) \approx C(\mathbf{p}) + (\nabla C(\mathbf{p})) \cdot \delta\mathbf{p}, \tag{13}$$

where the gradient operator ∇ acts over each of the components of $\mathbf{p}$ (i.e. over each weight and bias in the network),

$$\nabla = \left(\frac{\partial}{\partial p_1}, \frac{\partial}{\partial p_2}, \dots, \frac{\partial}{\partial p_M} \right) = \left(\frac{\partial}{\partial w_{11}^{[2]}}, \dots, \frac{\partial}{\partial w_{n_{L-1} n_L}^{[L]}}, \frac{\partial}{\partial b_1^{[2]}}, \dots, \frac{\partial}{\partial b_{n_L}^{[L]}} \right) \tag{14}$$

To minimise the new weight $C(\mathbf{p} + \delta\mathbf{p})$ we wish to make the final term of (13) as negative as possible. The Cauchy-Schwartz inequality states that

$$|\,(\nabla C(\mathbf{p})) \cdot \delta\mathbf{p}| \le \|\nabla C(\mathbf{p})\|_2 \|\delta\mathbf{p}\|_2, \tag{15}$$

with equality only when $\nabla C(\mathbf{p})$ lies in the same direction as (is a multiple of) $\delta\mathbf{p}$. This suggests that to minimise $C(\mathbf{p} + \delta\mathbf{p})$ we should choose $\delta\mathbf{p}$ in the direction of $-\nabla C(p)$, and so we take

$$\delta\mathbf{p} = -\eta \nabla C(\mathbf{p}), \tag{16}$$

where the positive constant η is the *learning rate.* The training of the neural network therefore starts by choosing some initial parameters $\mathbf{p}$ for the network (we will choose these randomly), and repeatedly performing the iteration

$$\mathbf{p} \leftarrow \mathbf{p} + \delta\mathbf{p} = \mathbf{p} - \eta \nabla C(\mathbf{p}). \tag{17}$$

We would like this iteration to approach the global minimum of $C(\mathbf{p})$ within a reasonable number of steps. Unfortunately, minimisation of a nonlinear function in high dimensions (recall that even in our simple network $\mathbf{p}$ has 37 components) is a fundamentally difficult problem. For sufficiently small η, the Taylor series approximation (13) will be valid and each iteration of (17) will decrease the cost function. However, for this to be true in general, η has be arbitrarily small. In practice, setting η to a very small value means that that $\mathbf{p}$ changes only very slightly at each iteration, and a very large number of steps is needed to converge to a minimum. On the other hand, setting η too large means that the approximation (13) is rarely valid and the choice of step (16) will not decrease the cost function as desired. Choosing an appropriate learning rate η is a balance between these two considerations.

Even if the iteration converges, it may converge to a local minimum of C, where $\nabla C = 0$, but where C is nonetheless much larger than the global minimum cost $\min_{\mathbf{p}}(C(\mathbf{p}))$. Finding the global minimum of an arbitrary nonlinear function in a high-dimensional space is near impossible. We therefore abandon the goal of finding the global minimum of $C(\mathbf{p})$, and instead simply look for a value of $\mathbf{p}$ that has a cost $C(\mathbf{p})$ less than some small threshold, τ, say.

2.3.2 Stochastic gradient descent

Each step of the steepest descent iteration (17) requires an evaluation of $\nabla C(\mathbf{p})$ which contains the derivatives of the cost function with respect to each of the weights and biases in the network. Recalling the definition of the cost function (9), we write

$$\nabla C(\mathbf{p}) = \frac{1}{N}\sum_{i=1}^{N} \nabla C_{\mathbf{x}^{\{i\}}}(\mathbf{p}) \tag{18}$$

where the contribution to the cost from each of the training data points i is

$$C_{\mathbf{x}^{\{i\}}}(\mathbf{p}) = \frac{1}{2}\|\mathbf{y}^{\{i\}} - \mathbf{a}^{[L]}(\mathbf{x}^{\{i\}})\|_2^2. \tag{19}$$

Evaluation of the gradient of the cost (18) can be computationally expensive, since the number of points N in the training data set may be large (many tens of thousands), and the number of elements in ∇C (the number of parameters in the weights and biases) may be several million in a large neural network. This means that it takes a long time to calculate each iteration of the steepest descent method (17).

A faster alternative is to instead calculate the increment $\delta\mathbf{p}$ at each iteration from the gradient of cost associated with a single randomly chosen training point,

$$\mathbf{p} \leftarrow \mathbf{p} + \delta\mathbf{p} = \mathbf{p} - \eta\nabla C_{\mathbf{x}^{\{i\}}}(\mathbf{p}), \tag{20}$$

a technique called stochastic gradient descent, or simply stochastic gradient. In stochastic gradient the reduction in the total cost function at (9) is likely to be smaller than in the steepest descent method (17) – in fact many steps are likely to increase the total cost slightly – but since many more iterations can be performed in a given time, the convergence is often quicker.

There are several ways of choosing the training data point i to be used at each step, but we will use the simplest: at each step randomly select (with replacement) a training data point i independently of previous selections.

2.3.3 Evaluating $\nabla C_{\mathbf{x}^{\{i\}}}(\mathbf{p})$: Back-propagation

To use the stochastic gradient algorithm (20) we must evaluate $\nabla C_{\mathbf{x}^{\{i\}}}(\mathbf{p})$, the derivative of the cost function associated with a single training data point with respect to each of the network weights and biases,

$$\frac{\partial C_{\mathbf{x}^{\{i\}}}}{\partial b_j^{[l]}} \qquad \text{for} \quad l = 2, \ldots, L, \tag{21}$$

$$\frac{\partial C_{\mathbf{x}^{\{i\}}}}{\partial w_{jk}^{[l]}} \qquad \text{for} \quad l = 2, \ldots, L. \tag{22}$$

The dependence of the cost $C_{\mathbf{x}^{\{i\}}}$ on the weights and biases seems rather complicated, so an obvious temptation is to evaluate the derivatives by finite-differencing. This is possible, easy to implement and highly recommended for validation/debugging purposes (see below). However the approach is also extremely expensive[3] and not viable for practical computations with big neural networks.

It turns out that the recursive nature of the network helps to simplify the analytical (and efficient) calculation of the required derivatives through an algorithm called *back-propagation*. Details can be found in the paper by Higham and Higham (2019). The key quantities required are known (for somewhat obscure reasons) as the *errors*, defined as

$$\delta_j^{[l]} = \frac{\partial C_{\mathbf{x}^{\{i\}}}}{\partial z_j^{[l]}} \qquad \text{for } 1 \le j \le n_l \quad \text{and} \quad 2 \le l \le L. \tag{23}$$

They represents the derivative of the cost for our chosen training point i with respect to the input z to the jth neuron at layer l.

Higham and Higham (2019) show that the errors can be computed by moving backwards through the network, starting with the output layer, $l = L$, for which

$$\boldsymbol{\delta}^{[L]} = \sigma_L'(\mathbf{z}^{[L]}) \circ \left(\mathbf{a}^{[L]} - \mathbf{y}^{\{i\}}\right), \tag{24}$$

where the operator $\circ$ is the componentwise (Hadamard) product, defined by

$$\mathbf{u} \circ \mathbf{v} = (u_1 v_1, u_2 v_2, \ldots, u_n v_n) \quad \text{for } \mathbf{u}, \mathbf{v} \in \mathbb{R}^n. \tag{25}$$

The Hadamard product can be avoided by writing equation (24) using a product with a diagonal matrix,

$$\boldsymbol{\delta}^{[L]} = \begin{pmatrix} \sigma_L'(z_1^{[L]}) & & & \\ & \sigma_L'(z_2^{[L]}) & & \\ & & \ddots & \\ & & & \sigma_L'(z_{n_L}^{[L]}) \end{pmatrix} \left(\mathbf{a}^{[L]} - \mathbf{y}^{\{i\}}\right). \tag{26}$$

Having established the errors for the output layer ($l = L$), we then move backwards through the network using the recurrence

$$\boldsymbol{\delta}^{[l]} = \sigma_l'(\mathbf{z}^{[l]}) \circ (\mathbf{W}^{[l+1]})^T \boldsymbol{\delta}^{[l+1]} \qquad \text{for} \quad l = L-1 \ldots, 2, \tag{27}$$

[3]No such thing as a free lunch!

which can again be written as

$$\boldsymbol{\delta}^{[l]} = \begin{pmatrix} \sigma_l'(z_1^{[l]}) & & & \\ & \sigma_l'(z_2^{[l]}) & & \\ & & \ddots & \\ & & & \sigma_l'(z_{n_l}^{[l]}) \end{pmatrix} (\mathbf{W}^{[l+1]})^T \boldsymbol{\delta}^{[l+1]} \qquad \text{for} \quad l = L-1, \dots, 2. \tag{28}$$

Note how this produces the errors in layer l from those already computed in layer $l+1$.

The real punchline is that the derivatives (21) and (22) can be expressed in terms of the the errors as

$$\frac{\partial C_{\mathbf{x}^{\{i\}}}}{\partial b_j^{[l]}} = \delta_j^{[l]} \qquad \text{for} \quad l = 2, \dots, L, \tag{29}$$

$$\frac{\partial C_{\mathbf{x}^{\{i\}}}}{\partial w_{jk}^{[l]}} = \delta_j^{[l]} a_k^{[l-1]} \qquad \text{for} \quad l = 2, \dots, L. \tag{30}$$

This is not only very neat (much neater than one may have expected, given the complicated structure of the network) but also much more efficient than the evaluation of the derivatives by finite-differencing.

Computing the gradients by back-propagation for a given pair of input and target outputs $(\mathbf{x}^{\{i\}}, \mathbf{y}^{\{i\}})$ from our training data then works as follows:

Algorithm 1 Computing the gradients by back-propagation

1. Apply the feed-forward algorithm by setting
$$\mathbf{a}^{[1]} = \mathbf{x}^{\{i\}} \tag{31}$$
and then recursively computing
$$\mathbf{z}^{[l]} = \mathbf{W}^{[l]}\mathbf{a}^{[l-1]} + \mathbf{b}^{[l]} \tag{32}$$
$$\mathbf{a}^{[l]} = \sigma_l\left(\mathbf{z}^{[l]}\right) \tag{33}$$
for $l = 2, 3, \ldots, L$. Make sure you store the intermediate results $\mathbf{z}^{[l]}$.

2. Using the output $\mathbf{a}^{[L]}$, the stored $\mathbf{z}^{[L]}$ and the target output $\mathbf{y}^{\{i\}}$, compute $\boldsymbol{\delta}^{[L]}$ from
$$\boldsymbol{\delta}^{[L]} = \sigma'_L(\mathbf{z}^{[L]}) \circ \left(\mathbf{a}^{[L]} - \mathbf{y}^{\{i\}}\right) \tag{34}$$
or use the Hadamard-free equivalent (26).

3. Then move backwards through the network to compute the errors in the other layers, using
$$\boldsymbol{\delta}^{[l]} = \sigma'_l(\mathbf{z}^{[l]}) \circ (\mathbf{W}^{[l+1]})^T \boldsymbol{\delta}^{[l+1]} \qquad \text{for} \quad l = L-1\ldots, \tag{35}$$
where you may again prefer to use the Hadamard-free equivalent (28). Note that $\boldsymbol{\delta}^{[l+1]}$ is available from the previous step and the $\mathbf{z}^{[l]}$ were stored during the feed-forward computation.

4. Now compute the required derivatives from
$$\frac{\partial C_{\mathbf{x}^{\{i\}}}}{\partial b_j^{[l]}} = \delta_j^{[l]} \qquad \text{for} \quad l = 2, \ldots, L, \tag{36}$$
$$\frac{\partial C_{\mathbf{x}^{\{i\}}}}{\partial w_{jk}^{[l]}} = \delta_j^{[l]} a_k^{[l-1]} \qquad \text{for} \quad l = 2, \ldots, L. \tag{37}$$

5. Done!

2.3.4 The training algorithm

We can now put everything together to describe the procedure for training a neural network using the stochastic gradient method:

Algorithm 2 The training algorithm

1. Choose a learning rate η, a target cost τ and a maximum number of iterations, `max_iter`. We will deem the network to be fully trained if the weights and biases have been adjusted such that the total cost for our training set, defined in equation (9), is less than τ.
2. Initialise each weight and bias of the network to a random value.
3. Initialise an iteration counter, `iter`=0.
4. Check if `iter` < `max_iter`, otherwise bail out and decree that the training has failed.
5. Choose a random training data point $i \in \{1, 2, \dots, N\}$.
6. Compute the derivatives $\partial C_{\mathbf{x}^{\{i\}}}/\partial b_j^{[l]}$ and $\partial C_{\mathbf{x}^{\{i\}}}/\partial w_{jk}^{[l]}$ either by finite-differencing or by using the back-propagation algorithm.
7. Update the weights and biases

$$b_j^{[l]} \leftarrow b_j^{[l]} - \eta \frac{\partial C_{\mathbf{x}^{\{i\}}}}{\partial b_j^{[l]}} \tag{38}$$

   and

$$w_{jk}^{[l]} \leftarrow w_{jk}^{[l]} - \eta \frac{\partial C_{\mathbf{x}^{\{i\}}}}{\partial w_{jk}^{[l]}} \tag{39}$$

8. Calculate the total cost C using equation (9). If $C < \tau$ the network is trained and we stop the algorithm.

   Note: The evaluation of the total cost C can be relatively expensive, particularly when N is large. It therefore makes sense to perform this convergence check only after every 1000 updates, say.
9. Increase the iteration counter, `iter++`, and return to step 4.

If the training fails you may wish to check if the most recent values of the weights and biases are actually good enough for your specific needs[4]. If not, try again with different choices for the learning rate, the initial weights and biases, the maximum number of iterations, etc. It's all a bit of black magic!

[4]Coming up with a sensible value for the target cost τ is difficult. From a mathematical point of view smaller is, of course, better, but this is obviously counteracted by the increasing computational cost. For a completely new problem it is not easy to predict a priori how the size of τ is related to the "quality" of the network (or even how to formally assess the latter). Trial and error (and experience!) are your best friends, as is reading up on heuristics that others have developed for their problems.

3 Coding tasks

The overall task is to create an objected-oriented implementation of a neural network and to explore its performance for a number of binary classification problems of the type discussed in section 1.

3.1 Provided code

The details of the design, implementation and validation are up to you, but to be able to test your code we obviously have to insist on some well-defined interfaces via which we can interact with it. For this purpose we provide the file `project2_a_basics.h` (best analysed with `doxygen`, of course) which provides:

- A base class `ActivationFunction` from which specific activation functions must be derived. It defines the pure virtual interface to the activation function $\sigma(x)$, and provides a finite-difference based default implementation of the derivative, for convenience.

- A sample activation function `TanhActivationFunction` that implements $\sigma(x) = \tanh(x)$ and its derivative.

- A base class `NeuralNetworkBasis` that you must use as the base class for your actual `NeuralNetwork` class. The `NeuralNetworkBasis` contains:

  - The pure virtual functions that you must implement in your own derived class:

    * `feed_forward(...)`: A function that computes the output from the final layer, $\mathbf{a}^{[L]}$, given the input $\mathbf{x}$ into the first layer, as described by equations (7) and (8).
    * `cost(...)`: A function that computes the cost (19) for a given single input $\mathbf{x}$ and the corresponding single target output $\mathbf{y}$.
    * `cost_for_training_data(...)`: A function that computes the total cost (9) for a given set of training data, comprising a collection of N inputs $\mathbf{x}^{\{i\}}$ and associated target outputs $\mathbf{y}^{\{i\}}$ $(i = 1, \ldots, N)$.
    * Two related functions,

      `read_parameters_from_disk(...)`

      and

      `write_parameters_to_disk(...)`

      that read/write the networks's data (weights, biases and type of activation function for each layer) from/to a file. The format of the file is explained below together with an example.
    * `train(...)`: A function that trains the network, given a set of training data (comprising a collection of N inputs $\mathbf{x}^{\{i\}}$ and associated target outputs $\mathbf{y}^{\{i\}}$ $(i = 1, \ldots, N)$), the learning rate η, the target cost τ, and the maximum number of iterations `max_iter`. It must also provide the option to document the progress of the stochastic gradient descent iterations in a file.

  - A (deliberately broken) virtual function that you must override in your derived class to make it work with whatever data structure you decide to use:

    * `initialise_parameters(...)`: A function that assigns normally distributed random values to the weights and biases. Since you decide how to store these, I can't implement this function for you[5]! However, the function shows you how to draw random numbers from a normal distribution with specified mean and standard deviation. Please recycle this in your own function.

There are also some fully-implemented helper functions that you are welcome to use:

- `read_training_data(...)`: A function that facilitates reading in training data from a file. The in-code comments explain the required format of the data, and two sample files `project_training_data.dat` and `spiral_training_data.dat` are provided on the course's Blackboard page.
- Two versions of the `output_training_data(...)` function which outputs the training data to a file in the format described in the in-code comments. One version of the function allows the specification of a filename, the other one takes an output stream.
- Two versions of the `output(...)` function which outputs the output[6] from the trained network, for a given set of input data. One version of the function allows the specification of a filename, the other one takes an output stream.

The file also provides a namespace `RandomNumber` that sets up a random number generator that you should use for (i) the initialisation of the weights and biases with random data, and (ii) the selection of training data during the stochastic gradient descent. You will note that the random number generator is seeded with a constant (12345) to make sure the runs are repeatable. Do not change this!

We also provide the tried and tested `dense_linear_algebra.h` file that you're familiar with from Project 1. It is already `#include`d in the `project2_a_basics.h` header file and you should use the `DoubleVector` and `DoubleMatrix` classes to represent the biases and weights, as well as the input and output vectors.

3.2 Suggested steps in the code development

Develop your `NeuralNetwork` class (and any other classes or functions that you may wish to introduce) in a header file `project2_a.h` that you will later upload to `gradescope`. The header file should `#include` the `project2_a_basics.h` file provided on the course's Blackboard page. Write your own driver code that includes your header file, and use it to test any newly created functionality. Also use it to work on the analysis exercises described below.

Here's a suggested sequence of steps for the code development:

1. Define the `NeuralNetwork` class and initially provide dummy implementations for the pure virtual functions defined in the `NeuralNetworkBasis` class to make sure the code compiles. The body of the dummy functions should simply issue a warning message saying that the function isn't implemented yet, and then abort.

[5] Otherwise I would. Honest!
[6] Yes, really. Does anybody know a synonym for output?

2. Design an object-oriented (!) data structure to represent the different layers, the associated data, and the way they interact. Fig. 4 sketches a possible design that worked for me, but feel free to invent your own.

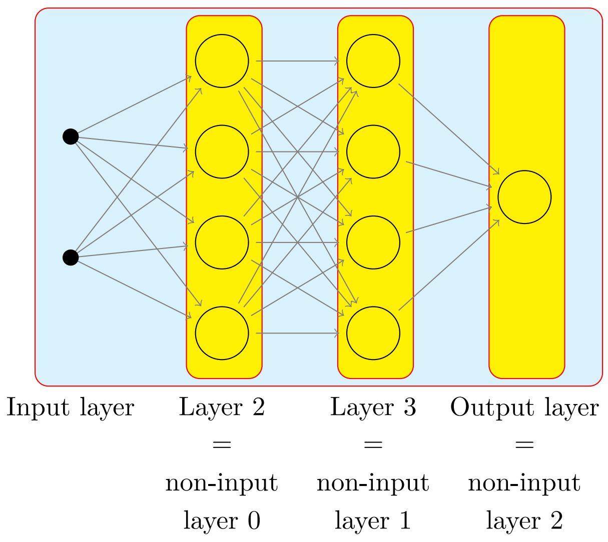
Input layer
Layer 2 = non-input layer 0
Layer 3 = non-input layer 1
Output layer = non-input layer 2

Figure 4: Suggested representation of a neural network in terms of two classes. The `NeuralNetwork` *class (cyan) comprises (stores) multiple* `NeuralNetworkLayer`*s (yellow). They store their weight matrix and bias vector, as well as the activation function, to be used by all their neurons. The* `NeuralNetworkLayer` *provides a member function that takes an input vector (the output from the previous layer) and returns an output vector, computed by applying the layer's weights, biases and its activation function to the input.*
Note that this is just an idea that reflects how I implemented it (with details deliberately omitted). Feel free to come up with your own design, but make sure you explain the "big picture" in your report (and by annotating it nicely in the code, of course!).

3. Implement the constructor so that it creates the required data structure and initialises all weights and biases to zero. The constructor must have the following interface:

```
/// Constructor: Pass the number of inputs (i.e. the number of
/// neurons in the input layer), n_input, and a vector of pairs,
/// containing for each subsequent layer (incl. the output layer)
///  (i) the number of neurons in that layer
///  (ii) a pointer to the activation function to be used by
///       all neurons in that layer,
///  so:
///
///   non_input_layer[l].first  = number of neurons in non-input
///                               layer l
///
///   non_input_layer[l].second = pointer to activation function
///                                for  all neurons in non-input
///                                layer l
///
/// Here l=0,1,..., with l=0 corresponding to the first internal
/// (hidden) layer.
```

```
NeuralNetwork(
 const unsigned& n_input,
 const std::vector<std::pair<unsigned,ActivationFunction*>>&
 non_input_layer)
```

Note that we're simply passing the number of layers and neurons to the constructor. How you create the actual layers from this input is up to you! If you follow my design (sketched in Fig. 4) where each layer is represented by a `NeuralNetworkLayer` class, the constructor will have to create these.

Here's a fragment of code that shows how to create a network comprising an input layer of size 2, two internal (hidden) layers of size 3 and an output layer of size 1. The three non-input layers use the `TanhActivationFunction`:

```
// Unit tests
//===========
if (argc==1)
{

 // Instantiate an activation function (same for all layers)
 //---------------------------------------------------------
 ActivationFunction* activation_function_pt=
  new TanhActivationFunction;


 // Build the network: 2,3,3,1 neurons in the four layers
 //------------------------------------------------------

 // Number of neurons in the input layer
 unsigned n_input=2;

 // Storage for the non-input layers: combine the number of neurons
 // and the (pointer to the) activation function into a pair
 // and store one pair for each layer in a vector:
 std::vector<std::pair<unsigned,ActivationFunction*>>
   non_input_layer;

 // The first internal (hidden) layer has 3 neurons
 unsigned n_neuron=3;
 non_input_layer.push_back(std::make_pair(n_neuron,
                                          activation_function_pt));

 // The second internal (hidden) layer has 3 neurons too!
 n_neuron=3;
 non_input_layer.push_back(std::make_pair(n_neuron,
                                          activation_function_pt));

 // Here's the output layer: A single neuron
 n_neuron=1;
 non_input_layer.push_back(std::make_pair(n_neuron,
                                          activation_function_pt));
```

4. (Re-)implement the broken virtual function

   `NeuralNetworkBasis::initialise_parameters(...)`,

   so that it assigns random values to the weights and biases, drawing the random values from a normal distribution with specified mean and standard deviation. The broken code in the `NeuralNetworkBasis` base class shows you how to generate the random numbers; the specific code required to assign these to the weights and biases obviously depends on how you represent these within your own data structure.

   Note: You may also wish to create an additional function

   `NeuralNetwork::initialise_parameters_for_test()`,

   say, that assigns specific values for the weights and biases for validation/debugging purposes.

5. Implement the `feed_forward(...)` function and validate it by checking that for a given set of weights and biases (assigned, e.g. using your

   `NeuralNetwork::initialise_parameters_for_test()`

   function) and a given input, you obtain the expected output.

6. Implement the `cost(...)` function and validate it by checking that for a given set of weights and biases, and a given input and target output, you obtain the expected cost.

7. Implement the `cost_for_training_data(...)` function and validate it by checking that for a given set of weights and biases, and a given set of inputs and associated target outputs, $(\mathbf{x}^{\{i\}}, \mathbf{y}^{\{i\}})$ $(i = 1, ..., N)$, you obtain the expected total cost.

8. Implement the `read_parameters_from_disk(...)` and `write_parameters_to_disk(...)` functions that will allow you to recover/record the parameters of a trained network. The file must contain for each non-input layer:

   - The name of the activation function (accessible from `ActivationFunction::name()`).
   - The dimension m of the input into the layer (i.e. the number of neurons in the previous layer).
   - The number of neurons n in the present layer.
   - The entries of the bias vector, recorded using n lines, each containing two numbers, $j \ b_j$, (no comma between them!) for $j = 0, ..., (n-1)$. **Note:** Assuming you represent the bias vectors using the `DoubleVector` class from the `dense_linear_algebra.h` header file, you can use the `DoubleVector::read(...)` and `DoubleVector::output(...)` functions to read/write this data.
   - The entries of the weight matrix, recorded using $n \times m$ lines, each containing the three numbers, $i \ \ j \ \ a_{ij}$, (no commas between them!) for $i = 0, ..., (n-1); j = 0, ..., (m-1)$. **Note:** Assuming you represent the weight matrices using the `DoubleMatrix` class from the `dense_linear_algebra.h` header file, you can use the `DoubleMatrix::read(...)` and `DoubleMatrix::output(...)` functions to read/write this output.

Here is an example file, `project_test_data.dat`, available on the course's Blackboard page, for a network comprising an input layer of size 2, two internal (hidden) layers of size 3 and an output layer of size 1. The three non-input layers use the `TanhActivationFunction`:

```
TanhActivationFunction
2
3
0 -0.887878
1 -5.49748
2 4.33655
0 0 -0.529004
0 1 -0.999406
1 0 7.58012
1 1 4.58883
2 0 -9.05766
2 1 -2.76716
TanhActivationFunction
3
3
0 1.84394
1 1.1788
2 -0.0224634
0 0 -0.982281
0 1 -0.427116
0 2 4.67974
1 0 -0.725688
1 1 3.47482
1 2 -2.35406
2 0 -0.751958
2 1 -4.80931
2 2 2.72185
TanhActivationFunction
3
1
0 -0.0549851
0 0 2.77672
0 1 -2.80889
0 2 -2.5583
```

Make sure your implementation of the `read_parameters_from_disk(...)` function contains some sanity checks to assert that the read-in data is consistent with the neural network it's supposed to be used for. Throw a runtime error if there are any inconsistencies.

9. Implement the `train(...)` function that trains the network using the stochastic gradient descent. Start by implementing a function that computes the derivatives $\partial C_{\mathbf{x}^{\{i\}}}/\partial w_{jk}^{[l]}$ and $\partial C_{\mathbf{x}^{\{i\}}}/\partial b_j^{[l]}$ by finite-differencing, then implement the more efficient version based on back-propagation. Compare the results from the two versions and make sure they agree (to within a some small difference due to the approximate nature

of the finite difference-based computation). Then embed them into the actual training algorithm.

10. Done – you can now do things with your network!

4 Analysis Exercises

1. **A simple test case:** The file `project_training_data.dat` contains the training data shown Fig. 5: The blue/red symbols indicate a value of -1/+1, respectively, and

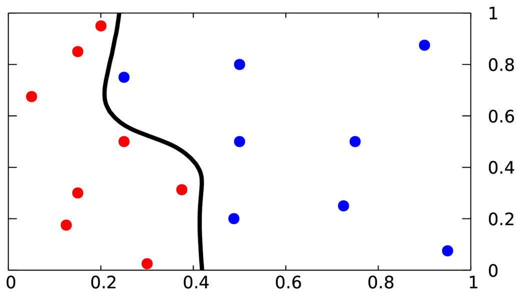

Figure 5: Plot of the training data in `project_training_data.dat`*. The black line shows a possible separation between the two regions within which the training data has values +1 (red) or -1 (blue)*

the black line shows a possible dividing line between the two regions.

- Create a network with $L = 4$ layers (one input layer, two hidden layers and one output layer), containing $(2, 3, 3, 1)$ neurons, respectively. Use the `TanhActivationFunction` for all layers. Train the network using a learning rate of $\eta = 0.1$ to a target cost of $\tau = 1 \times 10^{-4}$, starting with random initial values for the weights and biases, drawn from a normal distribution with zero mean and a standard deviation of 0.1. With these parameters the stochastic gradient descent should converge in fewer than 10^5 iterations.

  Document and describe the progress of the stochastic gradient descent by plotting the total cost as a function of the number of iterations (but, as mentioned above, don't recompute the cost after every iteration; it's too expensive! Doing it every every time you check for convergence should suffice to produce a useful graph – it's all the algorithm "sees" anyway). Assess the quality of the trained network using a plot similar to the one shown in Fig. 5, i.e. plot the training data with symbols and plot the zero line of the network's output computed on a regular grid of 100×100 plot points, say. (**Note:** There's a sample matlab code at the end of this document that shows you how to do this.)

- Repeat the training process four times and compare the output from the resulting networks. How reliable is the training process? How do you explain the discrepancies (if any)? (**Hint:** Make sure you perform the four training sessions in the same run, otherwise the random number generator will reset itself to the same seed and the four training sessions will be trivially identical. Here we want to start from different initial conditions to see if the stochastic gradient descent converges to the same solution every time.)

2. **A more challenging training set:** The file `spiral_training_data.dat` contains training data for a binary classification problem in which the red and blue points (again representing values +1 and -1, respectively) are located inside and outside a spiral region. This problem is more challenging and we suggest you use a learning

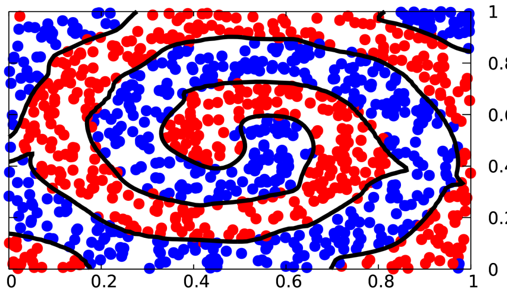

Figure 6: Plot of the training data in `spiral_training_data.dat`*. The black line shows a possible separation between the two regions.*

rate of $\eta = 0.01$ and try to train the network to a total cost below $\tau = 1 \times 10^{-3}$. Continue to use a normal distribution with zero mean and a standard deviation of 0.1 to initialise the network parameters. Allow for `max_iter` $= 4 \times 10^6$ iterations.

- **Network layout:**
  - Create a network with $L = 4$ layers (one input layer, two hidden layers and one output layer), containing $(2, 4, 4, 1)$ neurons, respectively. Use the `TanhActivationFunction` for all layers. Assess the convergence history and the quality of the network at the end of the training process. Note that the training process may not have reduced the total cost to the desired value of τ within the permitted number of iterations.
  - What happens if you increase the number of layers while keeping the number of neurons within the hidden layers constant? Try networks with $(2, 4, 4, 1)$, $(2, 4, 4, 4, 1)$ and $(2, 4, 4, 4, 4, 1)$ neurons, say. Does the increased computational cost result in a better performance of the network?
  - Repeat this exercise, this time increasing the number of neurons in the hidden layers while keeping the number of layers constant. Try networks with $(2, 4, 4, 1)$, $(2, 8, 8, 1)$ and $(2, 16, 16, 1)$ neurons, say.

Warning: Some of these runs will take a long time, so make sure you compile your code with full optimisation and disable range checking for the linear algebra code (so do not use `#define RANGE_CHECKING`). Compile with

```
g++ -O3 -o project_2a project_2a.cpp
```

(Note that the round thing in `-O3` is the letter O, not a zero!)

    - Which of these networks do you deem to give a satisfactory result? Specify how you defined “satisfactory” when reaching this verdict.

- For a network that produces a satisfactory outcome, document how the quality of the network improves throughout the training. You may wish to do this by plotting the output from the partially trained network (using plots similar to the one shown in Fig. 6) at regular intervals during the stochastic gradient descent.

3. **Computational cost:** Assess the cost of the training algorithm, and in particular, compare the cost of the two methods for computing the derivatives of the cost (finite differencing and back-propagation).

   For this purpose consider networks comprising N_{layer} layers, each with N_{neuron} neurons. Start by analysing (theoretically!) the cost of a single feed-forward operation, then analyse the two algorithms for computing the derivatives of the cost. You should find that all three operations (feed-forward, derivatives by finite-differencing and derivatives by back-propagation) have a computational cost that scales like $N_{\text{layer}}^a \times N_{\text{neuron}}^b$, for some (different) constants a and b. **Note:** You may assume N_{layer} and N_{neuron} to be large enough that tasks requiring N_{neuron} floating point operations can be neglected against those requiring N_{neuron}^2 operations, etc.

   Compare your theoretical predictions against actual timings from your code and discuss the level of agreement (or the lack thereof).

4. **A final inefficiency:** Our present implementation of the stochastic gradient descent algorithm (Algorithm 2) separates the computation of the derivatives of the cost (in step 6 of the algorithm) from the updates of the network's weights and biases (in step 7). While this is useful for code development and for validation purposes, it incurs an additional cost (in terms of memory and CPU time). Explain briefly how interlacing steps 6 and 7 would lead to a more efficient algorithm. Implement the improved algorithm and document the speedup achieved. **Hint:** Look at the derivation of the algorithm in Higham and Higham's (2019) paper, in particular their pseudo-code on page 873.

5 How to organise your code, what to upload, and how we will assess your work

What to upload

Your upload to GRADESCOPE must include the following files:

- The `dense_linear_algebra.h` and `project2_a_basics.h` header files from the course's BLACKBOARD page – even if you haven't changed them.

- Your own, newly-developed code in the form of a header file called `project2_a.h`. This ought to `#include` the `dense_linear_algebra.h` and `project2_a_basics.h` header files and any other C++ headers that your code needs. Your header file must provide the code for:

    - The fully-implemented `NeuralNetwork` class which must be derived from the `NeuralNetworkBasis` class defined in `project2_a_basics.h`.

  - Whatever other code you've written to make this work.

  Within GRADESCOPE, your header file will be `#include`d into a separate driver code and destruct tested, by checking that all the pure virtual functions have been implemented correctly.

- Your project report as a pdf file, called `project2_a.pdf`. Remember that this must be in "powerpoint" style, with a maximum of 20 pages, including the title page which must contain your student ID. A suggested (!) split for the number of subsequent pages (slides) is:

  - 5 pages for an overview of the overall task, and the structure of the code/classes you developed/implemented. No need to repeat the entire derivation presented in the notes, but provide enough information to make the overall scheme clear. [**3 marks**]
  - 2 pages to discuss the simple test case (Task 1 in section 4). [**5 marks**]
  - 4 pages to discuss the more challenging test case. (Spiral pattern; task 2 in section 4). [**5 marks**]
  - 5 pages for the theoretical analysis of the computational cost and the comparison with actual timings (Task 3 in section 4). [**5 marks**]
  - 2 pages for the analysis of how to improve the computational cost of the training process by interlacing the computation of the derivatives with the updates of the network's weights and biases (Task 4 in section 4). [**7 marks**]

- Also upload the C++ code that you wrote to perform the analysis summarised in your report as evidence that you've actually done the work – just in case there are any suspicions that the results presented in your report were generated by other means[7]. To avoid naming conflicts please prefix all additional files with "`my_`" (so call the driver code you've used for the analysis `my_project2_a.cpp` rather than `project2_a.cpp`, say).

How we'll assess your work

- We will use GRADESCOPE to assess the correctness of your code and check that exceptions are handled as required; there's no need to elaborate on your debugging, etc. in the report. [**20 marks**].

- We will award marks for code structure, clarity, layout and the provision of appropriate and consistent comments. [**10 marks**]

- When assessing the report we will award up to [**25 marks**] for the discussion of the specific issues discussed in section 4. Make sure you address all questions and provide evidence for any conclusions you draw from the computational data and/or your theoretical analysis. The breakdown of the marks is provided above.

  Readability/layout of the report and figures are also an important issue and are worth another [**5 marks**].

[7]You may have heard of CHATGPT...

References

Turing, A. (1950) Computing Machinery and Intelligence. Mind **59**, 433-460.

Higham, C.F. and Higham, D.J. (2019) Deep Learning: An introduction for Applied Mathematicians. SIAM Review **61** 860-891.

APPENDIX: Plotting level curves of functions in matlab/octave

In all our binary classification problems the input was two-dimensional (and could therefore be represented by points in the 2D plane) and the output was one-dimensional, meaning that the output of the trained network can be visualised as a function $z = f(x, y)$. Given that we wish to identify regions in which the output is (close to) +1 or -1, we can find the dividing line between these regions by plotting the zero level curve of the function $f(x, y)$, generated by evaluating the output from the network at a large number of (x, y) coordinates.

Given the output files written by our `NeuralNetwork` class, matlab or octave can produce plots that look roughly like those shown in Figs. 5 or 6. Creating contour plots in matlab is a bit of a pain, so here's a code that shows you how to do this:

```matlab
% Read data file
tData = readtable('output_network_2_8_8_1_run1.dat');

% Rename the columns of the table to 'x', 'y', and 'z' to
% represent the coordinates (x, y) and the value z.
tData.Properties.VariableNames = {'x','y','z'};

% Apply interpolation: Generate a rectangular grid
% for the x and y coordinates, ranging
% from 0 to 1 with a step size of 0.001.
[x1, y1] = meshgrid(0:0.001:1, 0:0.001:1);

% Perform grid interpolation to obtain the values of z
% at each point on the (x1, y1) grid.
z1 = griddata(tData.x, tData.y, tData.z, x1, y1);

% Show the result

% Create a contour plot, drawing a contour line at z = 0.
contour(x1, y1, z1, [0 0],'lineWidth', 1);

% Scale the x-axis and y-axis are equal.
axis equal

% Label the x-axis and y-axis.
xlabel('x')
ylabel('y')

% Add a title to the plot.
title('Contour plot')
```

Here `output_network_2_8_8_1_run1.dat` is a file written by

`NeuralNetworkBasis::output(...)`.

You can combine this with a scatter plot of the file written by

`NeuralNetworkBasis::output_training_data(...)`

You'll have to tweak things a bit to make the plots look as pretty as mine...