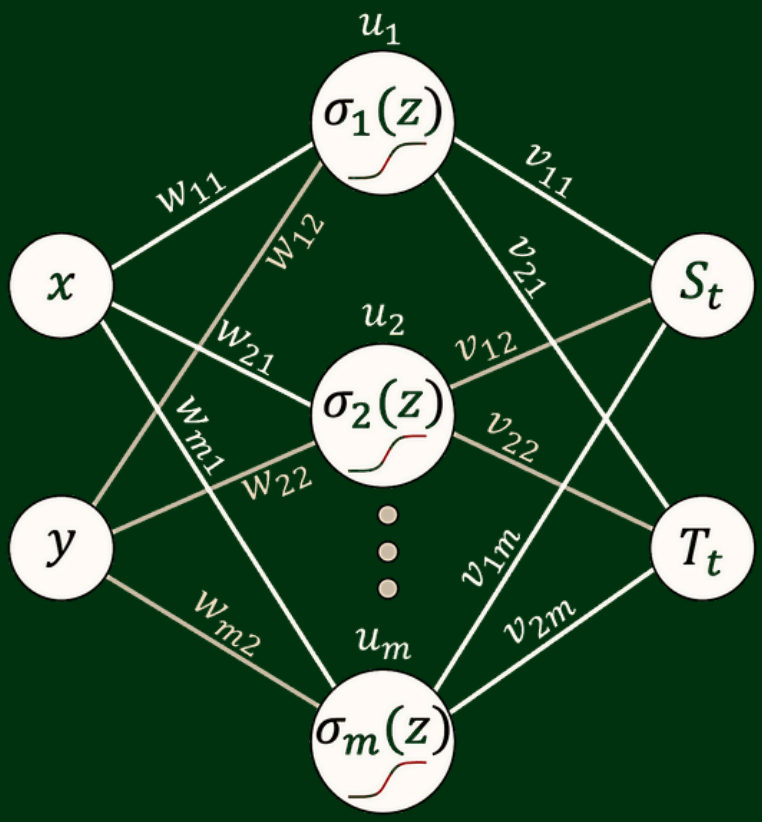
Modules for Machine and Deep Learning

Module 1: Fundamentals and Single-hidden Layer Network (with Matlab)



PM Ir. Dr. Airil Yasreen Mohd Yassin
School of Energy, Geoscience, Infrastructure and Society, Heriot-Watt University Malaysia

PM Dr. Ahmad Razin Zainal Abidin
Faculty of Civil Engineering, Universiti Teknologi Malaysia

Dr. Halinawati Hirol
Malaysia-Japan International Institute of Technology (MJIIT), Universiti Teknologi Malaysia

Dr. Mokhtazul Haizad Mokhtaram
Faculty of Engineering and Life Sciences, Universiti Selangor

Dr. Mohd Al-Akhbar Mohd Noor
Faculty of Engineering and Life Sciences, Universiti Selangor

CONTENTS

1. A very quick introduction

Machine Learning (ML) is a subset of Artificial Intelligence (AI). In turn, Deep Learning (DL) is a subset of ML. And today's very hot topics of Generative AI and Large Language Model (LLM) are subsets of DL. This is as shown in Figure 1 [1].

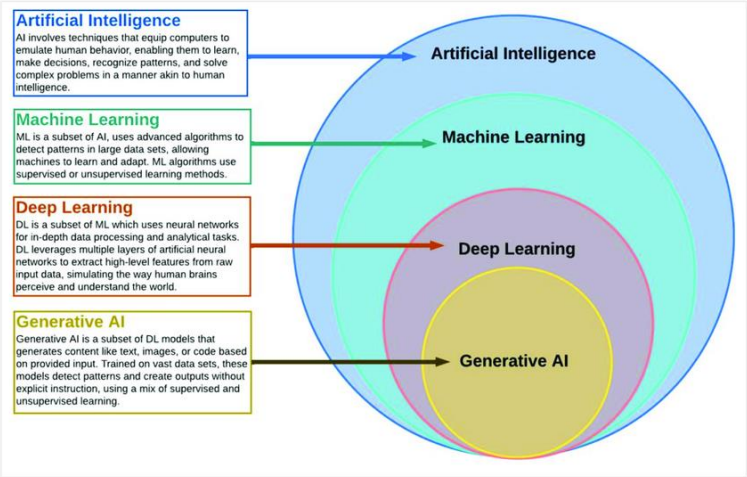


Figure 1: Specialization of deep learning (DL) in the field of AI [1]

DL is a type of ML that employs Artificial Neural-Network (ANN, or just NN) as its architecture.

Current technologies that use DL:

i) ChatGPT
ii) Google Translate
iii) Alpha Go
iv) many more

There are variants of DL, the three main ones are:

i) Feedforward Neural Networks (FFNNs)
ii) Recurrence Neural Networks (RNNs)
iii) Convolutional Neural Networks (CNNs)

While found itself in various applications, FFNNs also forms the core idea of RNNs and CNNs. For these reasons and being introductory, our focus in this workshop is FFNNs.

2. How are we going to conduct the workshop?

Answer:

We are going to jump straight to the action and discuss the details along the way.

Explanation:

The usual pedagogy is to discuss at the onset:

- definitions
- terminologies
- types
- advantages vs disadvantages
- etc.

which often times (in highly mathematical subjects) difficult to make much sense until "one gets his/her hands dirty".

This is our experience of more than two decades learning and teaching computational subjects like FEM, FDM, FVM etc. thus, ML and DL.

Therefore, in this workshop, a simultaneous equation is immediately given, and our task is to formulate a neural network (and deep neural network, if time permits) that able to produce "similar" results. As we progress with our derivation we will introduce and discuss specific features when it prompts explanation.

A learning process is effective this way for mathematical subjects. In other words,

"*Don't worry about the details first, just enjoy the maths and coding*".

3. The "unknown" equations to be mimic

In this workshop, we will create a network that is "equivalent" to the following equations:

$$3x + 2y = s \tag{1a}$$

$$2x + 6y = t \tag{1b}$$

or in matrix forms,

$$\begin{bmatrix} 3 & 2 \\ 2 & 6 \end{bmatrix} \begin{Bmatrix} x \\ y \end{Bmatrix} = \begin{Bmatrix} s \\ t \end{Bmatrix} \tag{1c}$$

But why we want to have these equations?

Answer:

We want to have these equations:

i. only for our workshop

ii. to create the sets of "labelled data", that is, the set of paired input (e.g. *s*, *t*) for the learning (e.g., training and testing) process of our network.

iii. to highlight that a network is actually just another mathematical function supposed to be equivalent to the unknown ones (having said this, we must pretend we don't know Eq (1) from now on, all we have is the labelled-data from activities to be exemplified in Figure 3)

Point ii. can be graphically described as in Figure 2.

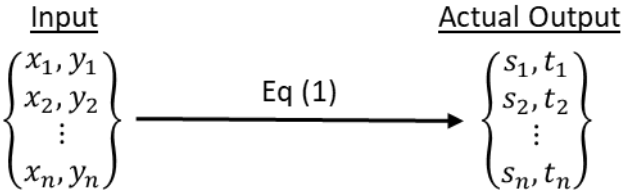


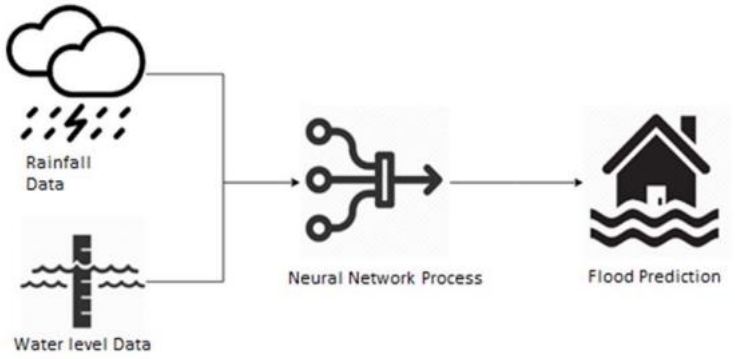
Figure 2: "Labelled data" from Eq (1)

In practice, labelled-data are obtained by other means such as physical measurements (i.e. gauges, sensors), grey-scale of pixels, surveys, etc. Unless we work with physics-informed machine learning (PiML), generally, there are no equations like Eqns. (1) for us to work with. In fact, it is the mission of DL to discover such a mathematical relationship between a set of paired known input and output data before predictions can be made.

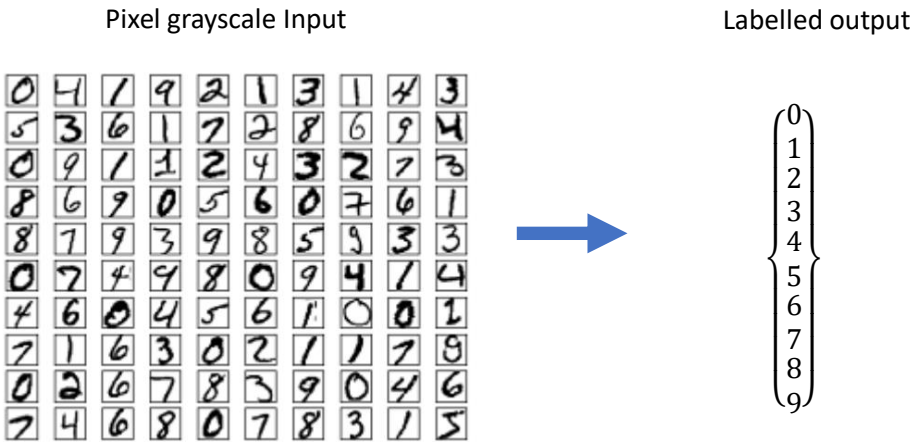
Figure 3 shows two examples of data gathering in research and/or practice. Figure 3(a) is an illustration of a flood prediction system. In hydrologic and flood modelling, we are required to predict the downstream flow for a given upstream condition. Usually, this involves the process of establishing the relationship between rainfall (or precipitation) represented by hyetograph and downstream flow represented by stage hydrograph. In ML,

the historical data of hyetograph and hydrograph are used as input and output, respectively.

Figure 3(b) shows the "Hello World" of DL, that is, the handwritten recognition problem. The greyscale of a screen where a number is written, is made input and the actual number is labelled as output. For a very good discussion, refer to Michael Nielson (http://neuralnetworksanddeeplearning.com). This is a go-to for those who wish to learn DL from scratch. Not only the explanation is easy to follow but it provides python codes and describes how you can access MNIST data set of 70,000 handwritten digit images for you to play around.



a) ANN flood prediction system [2]



b) Handwritten digit recognition [3]

Figure 3: Examples of data gathering activities

But it must be noted that, although we are using the data generated by Eqns. (1), it does not mean that our network and code are specific for the equations. Instead, our formulation and code will be general enough for them to be tweaked for practical use as in the cases depicted in Figure 3.

Such generality and the ability to "mimic" any functions, equations, and relationships, are what make the neural network a **universal approximator** (remember this term).

4. Single hidden layer (shallow) neural network

Deep learning is when a network has many so-called hidden layers, the more the deeper the learning. But let's start with a single hidden layer network. Even this would have the major components of deep learning already hence a good start.

To "mimic" Eqns. (1), we set a network below (Figure 4)

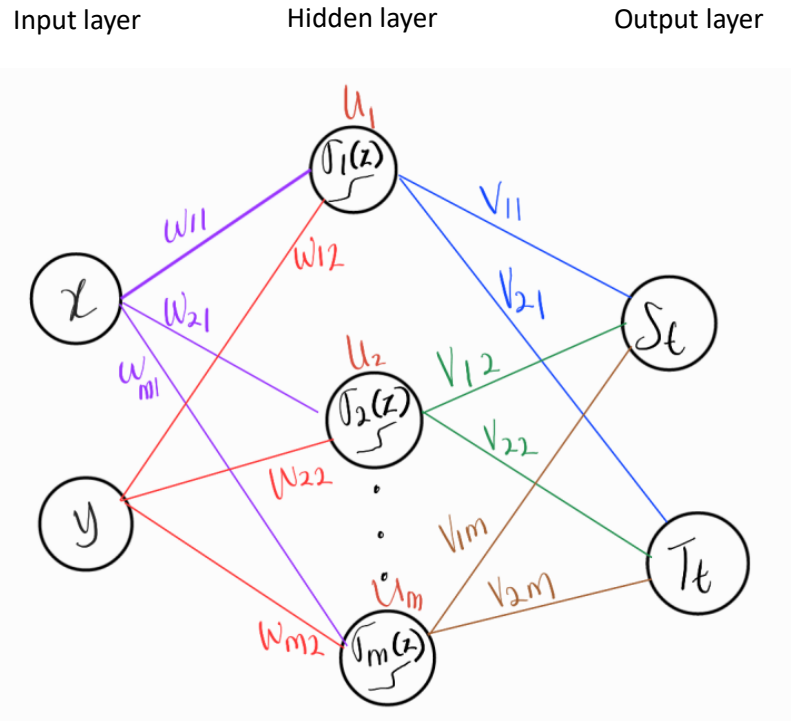


Figure 4: A single hidden layer (to "mimic" Eqns. (1))

Neurons

Except for circle circling input x and y, the circles symbolize (or called) neurons.

Input

x and y are the input variables. They enter the network as discrete (or evaluated) values. Thus, if we have n pairs of input, they can be arranged in array form such as,

$$\{\bar{x}\} = \begin{Bmatrix} x_k \\ y_k \end{Bmatrix} = \begin{Bmatrix} x_1 & x_2 & \dots & x_n \\ y_1 & y_2 & \dots & y_n \end{Bmatrix} \qquad (2^*)$$

Their layer (leftmost layer) is called input layer.

Channel

The lines connecting the circles (neurons) are called the channel. They guide the flow or the travel of "information" from one neuron to another.

Weights, w_{ji} and v_{ji}

Each channel is associated with a constant termed as weight. As we will see, the value of weights is the tuneable component of the network, the part that can be changed for the network to produce the desired results (or output). In other words, weights are the unknowns which first initialized (guessed) then iteratively tuned, a process famously known as training (remember this term).

Based on Figure 4, and if we have m neurons in the hidden layer, the weights can be arranged in vector or matrix forms (arrays) as follows,

$$[w] = \begin{bmatrix} w_{11} & w_{21} & \dots & w_{m1} \\ w_{12} & w_{22} & \dots & w_{m2} \end{bmatrix} \quad \begin{matrix} \text{Coming from} \\ \leftarrow x \\ \leftarrow y \end{matrix} \qquad (2)$$

$$[v] = \begin{bmatrix} v_{11} & v_{12} & \dots & v_{1m} \\ v_{21} & v_{22} & \dots & v_{2m} \end{bmatrix} \quad \begin{matrix} \text{Going to} \\ \rightarrow \hat{s} \\ \rightarrow \hat{t} \end{matrix} \qquad (3)$$

Biases, u_j

A bias is associated to a neuron. It acts as a threshold governing the "magnitude" of the output of a neuron. Historically, it governs the decision whether to activate (or to "fire") the neuron or not. Like the weights, biases are also the tuneable components of the network, the part that can be arranged for the network to produce the desired results (or output).

Based on Figure 4, and if we have m neurons in the hidden layer, the biases can be arranged in vector or matrix forms (arrays) as follows,

$$[u] = [u_1 \quad u_2 \quad \cdots \quad u_m] \tag{4}$$

Summing variable, z

As we will see in the discussion of forward pass, all input entering a neuron will be summed. As an example, let's consider the situation of the neuron below (Figure 5). As we can see, there are three inputs to the neuron, a, b, and c. They will be summed as

$$z = \mathrm{a} + \mathrm{b} + \mathrm{c} \tag{5}$$

*Note: We will show, what will be a, b, and c for our network in the discussion of forward pass.

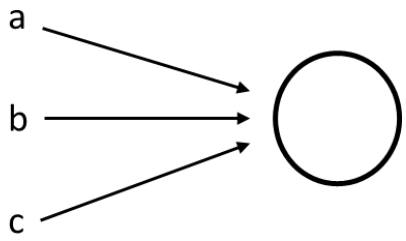


Figure 5: Input into neuron

The activation functions, σ_j

Activation function, σ_j is a possession of a neuron. Historically, activation function is simply the summing variable, z. But, in the modern neural network, activation function is formulated to add nonlinearity into the network. But why? Why we want to add nonlinearity to our network?

The answer is, because the relationship of observed data (between input and output) are generally nonlinear. There are various activation functions available but for now, we use sigmoid which is given as,

$$\sigma_j = \frac{1}{1 + e^{-z_j}} \tag{6}$$

Figure 6 shows a typical plot of a sigmoid function.

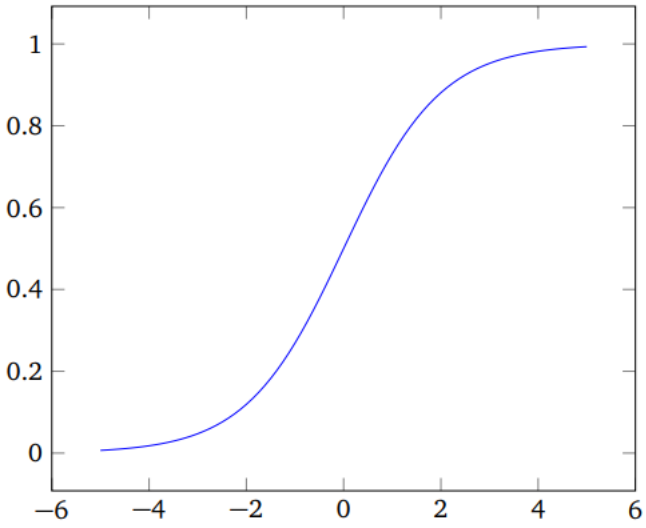


Figure 6: A typical plot of sigmoid function [3]

Again, based on Figure 4, and if we have m neurons in the hidden layer, the activation functions can be arranged in vector or matrix forms (arrays) as follows,

$$[\sigma] = [\sigma_1 \quad \sigma_2 \quad \cdots \quad \sigma_m] \tag{7a}$$

or

$$[\sigma] = \left[\frac{1}{1+e^{-z_1}} \quad \frac{1}{1+e^{-z_2}} \quad \cdots \quad \frac{1}{1+e^{-z_m}}\right] \tag{7b}$$

The network output, S_t and T_t

In Figure 4, S_t and T_t are the output and their neurons forms the output layer (the rightmost). What we want for these outputs to be as close as it can be to the correct ones. Specifically in our case to "mimic" Eq (1), we want S_t to be as close to s, and T_t to be as close to t. Once we able to do this, we can use the network to predict the output for the new or future input.

To get S_t and T_t to be as close as possible to s and t, respectively, we need to tune or train our weights and biases accordingly, a process we will detail later. Mathematically and by referring to Figure 4, we can state S_t and T_t as,

$$S_t = f(x, y, w, u, v) \tag{8a}$$

$$T_t = g(x, y, w, u, v) \tag{8b}$$

In words, we say, S_t and T_t are functions of the input, the weights, and the biases. Meaning, the values of S_t and T_t depend on the values of the input (x, y), the weights (w, v) and the biases (u).

We will derive the explicit expression of Eqns. (8) in our next discussion on forward pass.

5. Training the network (allowing it to "learn")

Basically, there are four stages in building or completing an ML or a DL project:

i. Training
ii. Validation
iii. Testing
iv. Prediction/discovery (the ultimate use)

Our historical or gathered data will be divided or separately stored and used for the first three stages (e.g., training, validation, and testing).

Training is the stage where we initialize the value of the parameters (e.g., weights and biases) then train or tune them iteratively (updating).

Validation would take place after the training. Validation involves the tuning of the hyperparameters such as learning rate, change of activation function, increase in the number of neurons and hidden layers etc. Validation might use the same sets of labelled data as in the training stage or different ones. The purpose of the validation is to check whether the performance of the network can be further optimized or be made more effective or/and economical.

Testing is the stage to confirm the performance or "correctness" of the network. It involves the use of the "unseen" data (or those never used before) to check whether the performance and the accuracy of the network is consistent, especially in avoiding overfitting. Overfitting is a situation where the network is very accurate for the data it trained for but not in making new prediction or discovery (bad at dealing with "unseen" or new data).

Since stage ii, iii, and iv, are more a matter of implementation (rather than fundamentals or basic concepts), in this workshop, we focus on stage i, that is, on the training of the network.

In turn, there are two stages for the training of the network:

i. Forward-pass (calculating the network output)
ii. Backward-pass (backpropagation and updating)

5.1 Forward Pass (calculating the network output)

A neural network is a computational graph (remember this term). It describes the flow and the process (mathematical operation) from input to output and from one node (neuron) to another. Specifically for the network of Figure 4, we can describe the flow and the process in 5 steps below.

Step 1: Multiplication

Input x_k will be multiplied by weight w_{11} before goes into the first neuron through the channel. It will also be multiplied by weight w_{21} before goes into the second neuron, and so on. For input y_k, it will be multiplied by weight w_{12} before goes into the first neuron and so on.

Step 2: Summation

But, before they enter the corresponding neuron, they will be summed first, then added by the corresponding bias. For example, for the 1st neuron, this will be

$$z_1 = w_{11}x_k + w_{12}y_k + u_1 \tag{9}$$

Whilst for the j^{th} neuron,

$$z_j = w_{j1}x_k + w_{j2}y_k + u_j \tag{10}$$

Take note that, Eqns. (9) and (10) are the explicit expression of Eqn. (5) for the network of Figure 4. Figure 7 shows graphically Eqn. (9), that is, just before it enters the first neuron.

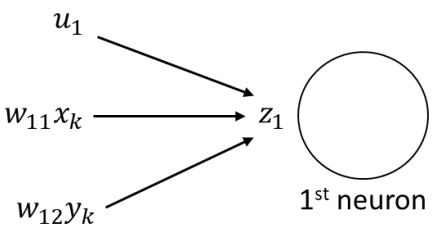


Figure 7: Input into the first neuron (for network of Fig. 4)

Note that, Figure 7 is Figure 5 for our specific network.

Step 3: Nonlinearization

In the neuron, the summed input (e.g. z) will be nonlinearized by the activation function. For the first neuron, this is given as (from Eqns. (7) and (9)),

$$\sigma_1 = \frac{1}{1+e^{-z_1}}$$

or

$$\sigma_1 = \frac{1}{1+e^{-(w_{11}x_k+w_{12}y_k+u_1)}} \tag{11}$$

For the j-th neuron (from Eqns. (7) and (10)),

$$\sigma_j = \frac{1}{1+e^{-z_j}}$$

or

$$\sigma_j = \frac{1}{1+e^{-(w_{j1}x_k+w_{j2}y_k+u_j)}} \tag{12}$$

Note that, Eqns. (11) and (12) are also the output of the neurons in the hidden layer. This is graphically described by Figure 8.

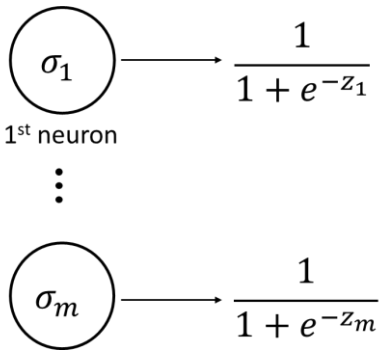


Figure 8: Output from hidden layer (before they get into the channel)

Step 4: Multiplication… again

Like step 1, before getting into the last layer, that is, the output layer, output of each neuron in the hidden layer will be multiplied by the weight of the corresponding channel. For example, for the output of the first neuron, it will be multiplied by weight, v_{11} before it gets to output neuron, S_t. It will also be multiplied by v_{21} before it gets to output neuron, T_t. And so on.

Step 5: Summation and the calculation of network output, S_t and T_t

Finally, all the info (variables) will reach the output layer. They will reach S_t and T_t, accordingly. Usually, there will be no more nonlinearization in the output neuron, only summation, thus, (refer to Fig. 4);

$$S_t = v_{11}\sigma_1 + v_{12}\sigma_2 \ldots + v_{1m}\sigma_m \tag{13}$$

$$T_t = v_{21}\sigma_1 + v_{22}\sigma_2 \ldots + v_{2m}\sigma_m \tag{14}$$

or (from Eqns. (11) and (12))

$$S_t = v_{11}\left(\frac{1}{1+e^{-(w_{11}x_k+w_{12}y_k+U_1)}}\right) + \ldots + v_{1m}\left(\frac{1}{1+e^{-(w_{m1}x_k+w_{m2}y_k+U_m)}}\right) \tag{15}$$

$$T_t = v_{21}\left(\frac{1}{1+e^{-(w_{11}x_k+w_{12}y_k+U_1)}}\right) + \ldots + v_{2m}\left(\frac{1}{1+e^{-(w_{m1}x_k+w_{m2}y_k+U_m)}}\right) \tag{16}$$

Now, we have calculated the network output through the forward pass!

Eqns. (13) and (14) (or (15) and (16)) are graphically given by Figure 9.

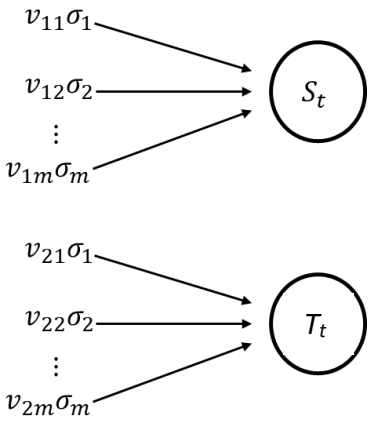


Figure 9: Calculation of the network output.

Take note that:

i. Eqns. (15) and (16) are the explicit expression of Eqns. (8a) and (8b), as promised.
ii. Eqns. (15) and (16) mimic Eqns. (1) as we mentioned earlier.
iii. Since Eqns. (15) and (16) are derived from our network of Figure 4, we should be seeing our network as mimicking Eq (1) too!

But at the beginning, S_t and T_t will not even be close to s and t. Especially when we initiate the process by guessing randomly the values of the weights and biases, in the process we called initialization (remember this term). There will be differences between s and S_t, and t and T_t, which we called errors (or residual errors). Eliminating these errors is what requires us to have the next stage, that is, the backward pass (or backpropagation).

5.2 Backward pass (backpropagation)

As mentioned, at the beginning (or before the network is fully trained), forward pass will generally produce errors. From the errors (or from its corresponding loss function) we will flow back into the network (from right to left) in calculating the gradients (all these will be discussed right after this) and to update the weights and biases so as to eliminate the error. This is under the scheme of gradient descent (GD) or its variants. This is the focus of this section, and we will start with the discussion of errors and loss functions.

5.2.1 Errors

The errors specific for our network against Eqns. (1) can be given as

$$E_1 = (s - S_t) \tag{17a}$$

$$E_2 = (t - T_t) \tag{17b}$$

or from Eqns. (13) and (14) as,

$$E_1 = (s - (v_{11}\sigma_1 + v_{12}\sigma_2 + \cdots + v_{1m}\sigma_m)) \tag{18a}$$

$$E_2 = (t - (v_{21}\sigma_1 + v_{22}\sigma_2 + \cdots + v_{2m}\sigma_m)) \tag{18b}$$

or from Eqns. (15) and (16) as,

$$E_1 = (s - (v_{11}(\frac{1}{1+e^{-(w_{11}x_k+w_{12}y_k+u_1)}}) + \cdots + v_{1m}(\frac{1}{1+e^{-(w_{m1}x_k+w_{m2}y_k+u_m)}}))) \tag{19a}$$

$$E_2 = (t - (v_{21}(\frac{1}{1+e^{-(w_{11}x_k+w_{12}y_k+u_1)}}) + \cdots + v_{2m}(\frac{1}{1+e^{-(w_{m1}x_k+w_{m2}y_k+u_m)}}))) \tag{19b}$$

It is long, isn't it? And it will get longer when we formulate the loss function. But don't worry, we will stick to the shorter form of Eqns. (17) for our future discussion. Also, we will employ an effective algorithm in handling future operation (in getting the gradients) which relies on chain-rule. Thus, the reasons I expanded Eqns. (17) into (18) and into (19) are for you to:

i. feel comfortable with the compactness of Eqns. (17)
ii. appreciate the need to insist on chain-rule (in dealing with the lengthy expression of the of the errors or the loss function). This is the spirit of auto-differentiation (remember this term).

5.2.2 Loss function and its minimization by gradient descent

As been repeated many times, the mission is to eliminate the errors of Eqns. (17). But the question is, how?

The answer is, we are going to resort to the approach of the calculus of variations. We are going to convert the errors of Eqns. (17) into a functional. Minimization of this functional would represent the elimination of the error.

In ML or DL, the functional is termed as **loss function (or cost function)**. There is various way to establish a loss function from the errors. In this workshop, we opt either Squared-Error or Mean-Squared Error (MSE), which both are given as

Squared-Error Loss Function,

$$\text{Loss function, } L = {E_1}^2 + {E_2}^2 \tag{20a}$$

$$L = (s - S_t)^2 + (t - T_t)^2 \tag{20b}$$

Mean-Squared

$$\text{Loss function, } L = \frac{1}{n}\sum_{k=1}^{n}({E_{1,k}}^2 + {E_{2,k}}^2) \tag{21a}$$

$$L = \frac{1}{n}\sum_{k=1}^{n}((s_k - S_{t,k})^2 + (t_k - T_{t,k})^2) \tag{21b}$$

where k describes the output calculated from the k-th pair of input (e.g. x_k, y_k) and n is the number of pair of input per batch or mini-batch (remember these terms) or per running.

For formulation discussion, we will opt for Eqns. (20) due to its simplicity but in our code, we opt for Eqns. (21). Reason being, Eqns. (21) suits batch or mini-batch treatment which is more robust.

It is also best to highlight that the use of Eqns. (20) means we employ Stochastic Gradient Descent (SGD) where we train for every input pair whilst Eqns. (21) is known as Batch or Mini-batch Gradient Descent where as mentioned, we train based on the average error of a batch or mini-batch.

Now, let's discuss how we can minimize the loss function (in eliminating the errors).

Figure 10 shows the hypothetical plot of the loss function assuming there is only one weight, say w_{11}. We need to make such an assumption because we cannot visualize the plot if there are many weights. The most we can visualize is two variables (say, two weights) which will be a surface plot. Let's just stick to one weight.

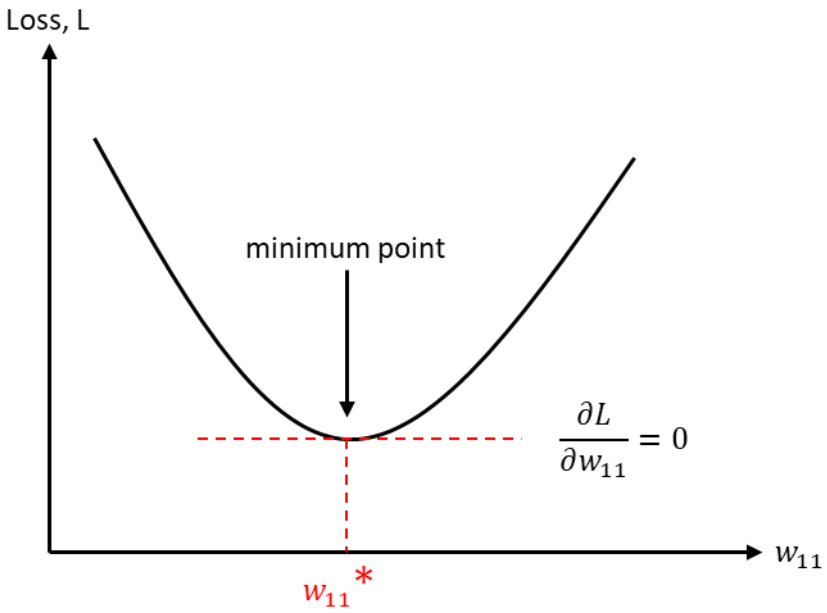


Figure 10: The hypothetical plot of loss function and the stationary principle

As we can see from Figure 10, the minimum of the loss function is at the location w_{11}*. Thus, w_{11}* is the desired value that eliminates the errors. But how can we determine w_{11}* if we are given only the function of the loss?

In stationary principle (as employed in various numerical techniques like finite element method), we can determine the value of w_{11}* by setting the first derivative of the loss function with respect to w_{11} to zero (e.g. $\frac{\partial L}{\partial w_{11}} = 0$). You should recall this from your basic calculus. However, we don't have such a luxury in ML. There would be tens if not hundreds of thousand of weights and biases making such a stationary principle too expensive if not impossible.

So, we resort to a different approach, that is the Gradient Descent (GD).

Figure 11 shows the idea of GD. In this scheme we rely on gradient to guide our way down the trough (the minimum point) of the loss function from an initial starting point.

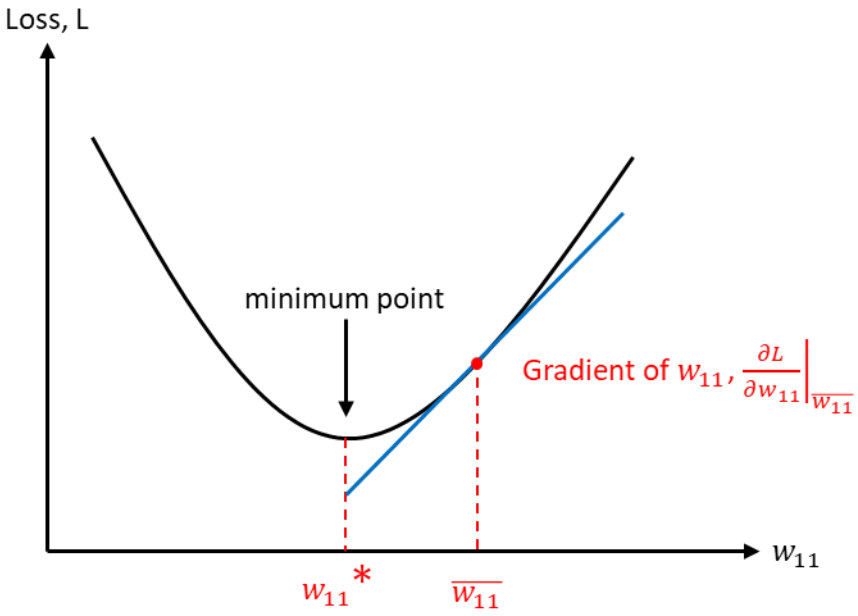


Figure 11: Searching minimum point by gradient descent

Let's take the starting point (obtained by randomly guessing or initializing the value of w_{11}) at $\overline{w_{11}}$. By taking the gradient at $\overline{w_{11}}$, the GD update rule is given as

$$w_{11}{}' = \overline{w_{11}} - \mu \frac{\partial L}{\partial w_{11}}\Big|_{\overline{w_{11}}} \tag{22}$$

where μ is a hyperparameter known as learning rate. We should discuss further about learning rate but for now, let's just say that its value usually in the range of 1×10^{-3} to 1×10^{-6}. $w_{11}{}'$ is the updated value of the weight.

To express it in a familiar form you might read elsewhere, we rewrite Eqn. (22) below,

$$w_{11}{}' = w_{11} - \mu \frac{\partial L}{\partial w_{11}} \tag{23}$$

Where we have omitted the overbar sign " $-$ " $from\ w_{11}$ and the evaluated sign "$|_{\overline{w_{11}}}$" from the gradient term. Eqn. (23) is only for weight, w_{11}. For the rest of weights and biases in our network, their update can be given as

$$w_{ji}{}' = w_{ji} - \mu \frac{\partial L}{\partial w_{ji}} \tag{24a}$$

$$u_j{}' = u_j - \mu \frac{\partial L}{\partial u_j} \tag{24b}$$

$$v_{ij}{}' = v_{ij} - \mu \frac{\partial L}{\partial v_{ij}} \tag{24c}$$

Eqns. (24) are what we need! They allow the parameters to be updated before entering the next iteration (repeatedly going through the forward-pass and the backward-pass until converged). This is the training of the network.

Now, we need to know how to calculate the gradient of the network with respect to the parameters (e.g., w_{ji}, u_j and v_{ij}).

As we will discuss next, it is most effective if we get the gradients by chain-rule.

5.2.3 Gradients with respect to the parameters by chain-rule

First, let's express our gradients (previously shown in Eqns. (24)) in terms of errors of Eqns. (20) for the convenient of our coding, thus

For v_{ij}

$$\frac{\partial L}{\partial v_{ij}} = 2\left(-E_1 \frac{\partial S_t}{\partial v_{ij}} - E_2 \frac{\partial T_t}{\partial v_{ij}}\right) \tag{25a}$$

since s and t will vanish during the differentiation.

For u_j

$$\frac{\partial L}{\partial u_j} = 2\left(-E_1 \frac{\partial S_t}{\partial u_j} - E_2 \frac{\partial T_t}{\partial u_j}\right) \tag{25b}$$

For w_{ji}

$$\frac{\partial L}{\partial w_{ji}} = 2\left(-E_1 \frac{\partial S_t}{\partial w_{ji}} - E_2 \frac{\partial T_t}{\partial w_{ji}}\right) \tag{25c}$$

Now, we will discuss how to get the six gradients in Eqns. (24) which are, $\frac{\partial s_t}{\partial v_{ij}}$, $\frac{\partial T_t}{\partial v_{ij}}$, $\frac{\partial s_t}{\partial u_j}$, $\frac{\partial T_t}{\partial u_j}$, $\frac{\partial s_t}{\partial w_{ji}}$, and $\frac{\partial T_t}{\partial w_{ji}}$, by chain-rule. But, why chain-rule? For the following reasons:

i. it is effective.
ii. to prepare you for future discussion on auto-differentiation or autograd (remember these terms) which chain-rule is key.

Chain-rule

Actually, we can get the gradients immediately by differentiating Eqns. (19). This is called analytical differentiation. But, as you can expect, if you have hundreds or thousands of neurons, it will be too long and expensive to differentiate analytically. Being a computational graph, our network provides clear relationship between variables thus we should exploit this aspect and coming out with an efficient framework for the differentiation process in getting the gradients.

If we observe, our network involves simple mathematical operations and variables, if they are kept separated (not implicitly combined). In terms of operations, we basically just have:

i. Summation
ii. Multiplication
iii. Nonlinearization (inserting variables into activation functions)

In terms of functions, we operate simple functions like linear functions or the easily differentiated **activation functions** like,

i. Sigmoid
ii. Tanh
iii. ReLU

Now let's do the separate differentiations to prepare for chain-rule later.

i. Differentiation of *z* (Eq (9) and (10))

Referring to Eqns. (9) and (10), for the *j*-th neuron, the summing variable, *z* can be given in indexed form as,

$$z_j = w_{j1}x_k + w_{j2}y_k + u_j \tag{26}$$

thus,

$$\frac{\partial z_j}{\partial w_{j1}} = x_k \tag{27a}$$

$$\frac{\partial z_j}{\partial w_{j2}} = y_k \tag{27b}$$

$$\frac{\partial z_j}{\partial u_j} = 1 \tag{27c}$$

ii. Differentiation of sigmoid, σ (Eqns. (11) and (12))

Referring to Eqns. (11) and (12), for the *j*-th neuron, the activation function, σ can be given in indexed form as,

$$\sigma_j = \frac{1}{1 + e^{-z_j}} \tag{28}$$

thus,

$$\frac{\partial \sigma_j}{\partial z_j} = \sigma_j(1 - \sigma_j) \tag{29}$$

iii. Differentiation of the output (S_t and T_t) (Eqns. (13) and (14))

Referring to Eqns. (13) and (14), the network output can be given in indexed form as,

$$S_t = v_{1j}\sigma_j \tag{30a}$$

$$T_t = v_{2j}\sigma_j \tag{30b}$$

thus,

$$\frac{\partial S_t}{\partial \sigma_j} = v_{1j} \tag{31a}$$

$$\frac{\partial T_t}{\partial \sigma_j} = v_{2j} \tag{31b}$$

Now, we are ready to employ the chain-rule to obtain our output gradients with respect to the parameters (weights and biases).

i. Output gradient with respect to w_{ij}

Employing chain-rule,

$$\frac{\partial S_t}{\partial w_{ij}} = \frac{\partial S_t}{\partial \sigma_j}\frac{\partial \sigma_j}{\partial z_j}\frac{\partial z_j}{\partial w_{ij}} \tag{32a}$$

$$\frac{\partial T_t}{\partial w_{ij}} = \frac{\partial T_t}{\partial \sigma_j}\frac{\partial \sigma_j}{\partial z_j}\frac{\partial z_j}{\partial w_{ij}} \tag{32b}$$

But, by referring to Eqns. (27a) and (27b), we have different results for $\frac{\partial z_j}{\partial w_{ij}}$ depending on w_{j1} or w_{j2}. It is therefore convenient to make separate gradients hence update for w_{j1} and w_{j2} (we adopted this in our code).

Thus, instead of using Eqn. (32a) for output S_t, we give the following

$$\frac{\partial S_t}{\partial w_{j1}} = \frac{\partial S_t}{\partial \sigma_j}\frac{\partial \sigma_j}{\partial z_j}\frac{\partial z_j}{\partial w_{j2}} \tag{33a}$$

$$\frac{\partial S_t}{\partial w_{j2}} = \frac{\partial S_t}{\partial \sigma_j}\frac{\partial \sigma_j}{\partial z_j}\frac{\partial z_j}{\partial w_{j2}} \tag{33b}$$

By inserting Eqns. (27), (29) and (31) into (33), we obtain,

$$\frac{\partial S_t}{\partial w_{j1}} = v_{1j}x_k\sigma_j(1 - \sigma_j) \tag{34a}$$

$$\frac{\partial S_t}{\partial w_{j2}} = v_{1j}y_k\sigma_j(1 - \sigma_j) \tag{34b}$$

Eqns. (34) are what you will find in our code!

Repeating the same process, the gradients for T_t can be given as,

$$\frac{\partial T_t}{\partial w_{j1}} = v_{2j} x_k \sigma_j (1 - \sigma_j) \tag{35a}$$

$$\frac{\partial T_t}{\partial w_{j2}} = v_{2j} y_k \sigma_j (1 - \sigma_j) \tag{35b}$$

Eqns. (34) and (35) are what you will find in our code!

ii. Output gradient with respect to u_j

Employing chain-rule,

$$\frac{\partial S_t}{\partial u_j} = \frac{\partial S_t}{\partial \sigma_j} \frac{\partial \sigma_j}{\partial z_j} \frac{\partial z_j}{\partial u_j} \tag{36a}$$

$$\frac{\partial T_t}{\partial u_j} = \frac{\partial T_t}{\partial \sigma_j} \frac{\partial \sigma_j}{\partial z_j} \frac{\partial z_j}{\partial u_j} \tag{36b}$$

By inserting Eqns. (27), (29) and (31) into (36), we obtain

$$\frac{\partial S_t}{\partial u_j} = v_{1j}(1)\sigma_j(1 - \sigma_j) \tag{37a}$$

$$\frac{\partial T_t}{\partial u_j} = v_{2j}(1)\sigma_j(1 - \sigma_j) \tag{37b}$$

Eqns. (37) are what you will find in out code!

iii. Output gradient with respect to v_{ij}

Based on Eqn. (30), we can see that we don't need chain-rule thus

$$\frac{\partial S_t}{\partial v_{1j}} = \sigma_j \tag{38a}$$

$$\frac{\partial T_t}{\partial v_{2j}} = \sigma_j \tag{38b}$$

Eqns. (38) are what you will find in our code!

So that's it, Eq (34), (35), (37) and (38) are all the gradients we need. Now, we can finalize the updating of our parameters.

5.3 Updating the network parameters (w_{ji}, u_j, v_{ij})

By inserting Eqns. (25) into (24) accordingly,

$$w'_{j1} = w_{j1} - 2\mu\left(-E_1\frac{\partial S_t}{\partial w_{j1}} - E_2\frac{\partial T_t}{\partial w_{j1}}\right) \tag{39a}$$

$$w'_{j2} = w_{j2} - 2\mu\left(-E_1\frac{\partial S_t}{\partial w_{j2}} - E_2\frac{\partial T_t}{\partial w_{j2}}\right) \tag{39b}$$

$$u'_j = u_j - 2\mu\left(-E_1\frac{\partial S_t}{\partial u_j} - E_2\frac{\partial T_t}{\partial u_j}\right) \tag{39c}$$

$$v'_{1j} = v_{1j} - 2\mu\left(-E_1\frac{\partial S_t}{\partial v_{1j}}\right) \tag{39d}$$

$$v'_{2j} = v_{2j} - 2\mu\left(-E_2\frac{\partial T_t}{\partial v_{2j}}\right) \tag{39e}$$

where,

E_1 and E_2 are given by Eqns. (17)

$\frac{\partial S_t}{\partial w_{j1}}$ and $\frac{\partial S_t}{\partial w_{j2}}$ are given by Eqns. (34)

$\frac{\partial T_t}{\partial w_{j1}}$ and $\frac{\partial T_t}{\partial w_{j2}}$ are given by Eqns. (35)

$\frac{\partial S_t}{\partial u_j}$ and $\frac{\partial T_t}{\partial u_j}$ are given by Eqns. (37)

$\frac{\partial S_t}{\partial v_{1j}}$ and $\frac{\partial T_t}{\partial v_{2j}}$ are given by Eqns. (38)

Why I described the updating the way I did above? Because that is how you will find it in our code. Now, we can go to the code. But before that, I want to summarize the training process in a flowchart.

5.4 The network training flowchart

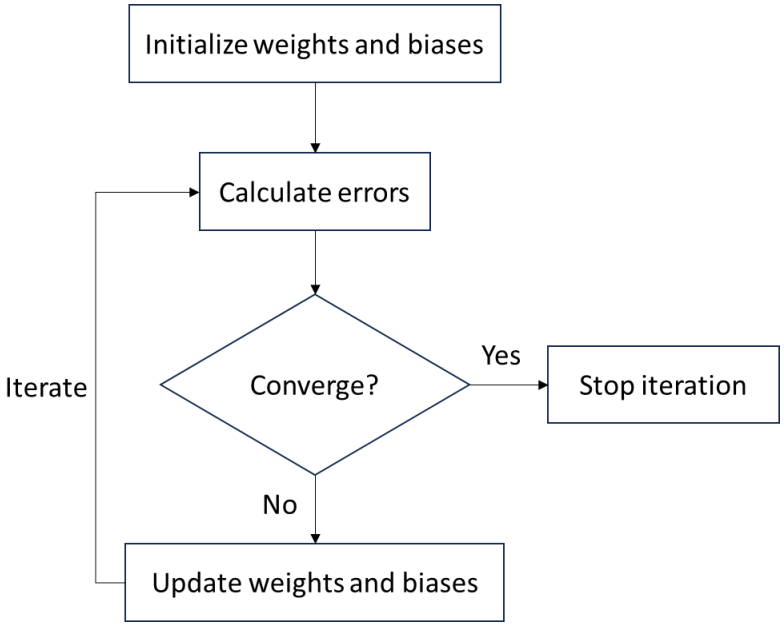


Figure 12 Flowchart of the training of network

6. Matlab code

Box 1 shows the script to generate and to initialize the weights and the biases. M is the number of neurons in the hidden layer which can freely be decided. To initialize the values randomly, we can use Matlab command "rand()" or "randi()". To ease the cracking of the script, the equations they coded are given beneath it.

The script

```
% STEP 1 : Generate inital Value for weight and biases
M = 2;
v = rand(2,M)-1/2;
u = rand(1,M)-1/2;
w = rand(2,M)-1/2;
```

The equations coded

$$[w] = \begin{bmatrix} w_{11} & w_{21} & \cdots & w_{m1} \\ w_{12} & w_{22} & \cdots & w_{m2} \end{bmatrix} \quad (2)$$

$$[v] = \begin{bmatrix} v_{11} & v_{12} & \cdots & v_{1m} \\ v_{21} & v_{22} & \cdots & v_{2m} \end{bmatrix} \quad (3)$$

$$[u] = [u_1 \quad u_2 \quad \cdots \quad u_m] \quad (4)$$

Box 1

Box 2 shows the generation of input where N is the number of input which can freely be decided. Again, we generate the input randomly so that we can have various output from Eqns. (1).

The script

```
xk = (round(rand(1,N)*L,3));
yk = (round(rand(1,N)*L,3));
```

The equations coded

$$\{\bar{x}\} = \begin{Bmatrix} x_k \\ y_k \end{Bmatrix} = \begin{Bmatrix} x_1 & x_2 & \cdots & x_n \\ y_1 & y_2 & \cdots & y_n \end{Bmatrix} \quad (2^*)$$

Box 2

Box 3 shows the scripts of the summing variables, z.

The script

```
z = [];
for j = 1:M %loop over neuron
    zj = w(1,j)*xk + w(2,j)*yk + u(j);
    z = cat(1,z,zj);
```

The equations coded

$$z_m = w_{m1}x_k + w_{m2}y_k + u_m \qquad (10)$$

Box 3

Box 4 shows the activation function and its derivative.

The script

```
sig = 1 ./ (1 + exp(-z));
dsigdz = sig .* (1-sig);
```

The equations coded

$$\sigma_m = \frac{1}{1 + e^{-z_m}} \qquad \text{(12a) or (28)}$$

$$\frac{\partial \sigma_j}{\partial z_j} = \sigma_j(1 - \sigma_j) \qquad (29)$$

Box 4

Box 5 shows the script to calculate (or generate) the network output.

The script

```
St = 0;
Tt = 0;
for j = 1:M %loop over neuron
    St = St + v(1,j) *sig(j,:);
    Tt = Tt + v(2,j) *sig(j,:);
end
```

The equations coded

$$S_t = v_{1j}\sigma_j \qquad (30a)$$

$$T_t = v_{2j}\sigma_j \qquad (30b)$$

Box 5

Box 6 shows the scripts of the equations to be "mimic" that is Eqns. (1). The script generates the actual output for the error calculation.

The script

```
% STEP 5 : Calculate the actual equations
s = c1*xk + c2*yk;
t = d1*xk + d2*yk;
```

The equations coded

$$3x + 2y = s \qquad (1a)$$

$$2x + 6y = t \qquad (1b)$$

Box 6

Box 7 is for error calculation

The script

```
% STEP 6 : Calculate the residual errors
E1 = (s - St);
E2 = (t - Tt);
```

The equations coded

$$E_1 = (s - S_t) \qquad (17a)$$

$$E_2 = (t - T_t) \qquad (17b)$$

Box 7

Box 8 shows the script for $\frac{\partial S_t}{\partial v_{1j}}$ and $\frac{\partial T_t}{\partial v_{2j}}$. Note the different sign. Our derivation placed the negative at Eqns. (25) and (39).

The script

```
% diff over v1
dStdv1 = -sig(j,:);

% diff over v2
dTtdv2 = -sig(j,:);
```

The equations coded

$$\frac{\partial S_t}{\partial v_{1j}} = \sigma_j \qquad (38a)$$

$$\frac{\partial T_t}{\partial v_{2j}} = \sigma_j \qquad (38b)$$

Box 8

Box 9 shows the script for $\frac{\partial S_t}{\partial u_j}$ and $\frac{\partial T_t}{\partial u_j}$. Note again the different sign. Our derivation placed the negative at Eqns. (25) and (39).

The script

```
% diff over u
dStdu = -v(1,j) *dsigdz(j,:);
dTtdu = -v(2,j) *dsigdz(j,:);
```

The equations coded

$$\frac{\partial S_t}{\partial u_j} = v_{1j}(1)\sigma_j(1-\sigma_j) \qquad (37a)$$

$$\frac{\partial T_t}{\partial u_j} = v_{2j}(1)\sigma_j(1-\sigma_j) \qquad (37b)$$

Box 9

Box 10 shows the script for $\frac{\partial S_t}{\partial w_{j1}}$, $\frac{\partial S_t}{\partial w_{j2}}$, $\frac{\partial T_t}{\partial w_{j1}}$, and $\frac{\partial T_t}{\partial w_{j2}}$. Similar mention regarding the sign.

The script

```
% diff over w1
dStdw1 = -v(1,j) *xk .*dsigdz(j,:);
dTtdw1 = -v(2,j) *xk .*dsigdz(j,:);

% diff over w2
dStdw2 = -v(1,j) *yk .*dsigdz(j,:);
dTtdw2 = -v(2,j) *yk .*dsigdz(j,:);
```

The equations coded

$$\frac{\partial S_t}{\partial w_{j1}} = v_{1j} x_k \sigma_j (1 - \sigma_j) \quad (34a)$$

$$\frac{\partial S_t}{\partial w_{j2}} = v_{1j} y_k \sigma_j (1 - \sigma_j) \quad (34b)$$

$$\frac{\partial T_t}{\partial w_{j1}} = v_{2j} x_k \sigma_j (1 - \sigma_j) \quad (35a)$$

$$\frac{\partial T_t}{\partial w_{j2}} = v_{2j} y_k \sigma_j (1 - \sigma_j) \quad (35b)$$

Box 10

Finally, Box 11 shows the update of the parameters (e.g., the weight and the biases). With regards to what seemed to be a swap of the subscript for w_{ji}, whilst we derived based on the convention where the first subscript refers to the neuron of the next layer, the code refers to the element's entry in the matrix. Referring to Eqn. (2), we can see that element denoted as w_{21} by our convention is read as w [1,2] by the code due to its entry/location in the matrix.

The script

```
% STEP 8 : Update the weight and biases
v(1,j) = v(1,j) - eta*sum(2*E1.*dStdv1);
v(2,j) = v(2,j) - eta*sum(2*E2.*dTtdv2);
u(j)   = u(j)   - eta*sum(2*E1.*dStdu  + 2*E2.*dTtdu );
w(1,j) = w(1,j) - eta*sum(2*E1.*dStdw1 + 2*E2.*dTtdw1);
w(2,j) = w(2,j) - eta*sum(2*E1.*dStdw2 + 2*E2.*dTtdw2);
```

The equations coded

$$w'_{j1} = w_{j1} - 2\mu\left(-E_1\frac{\partial S_t}{\partial w_{j1}} - E_2\frac{\partial T_t}{\partial w_{j1}}\right) \quad (39a)$$

$$w'_{j2} = w_{j2} - 2\mu\left(-E_1\frac{\partial S_t}{\partial w_{j2}} - E_2\frac{\partial T_t}{\partial w_{j2}}\right) \quad (39b)$$

$$u'_{j} = u_{j} - 2\mu\left(-E_1\frac{\partial S_t}{\partial u_{j}} - E_2\frac{\partial T_t}{\partial u_{j}}\right) \quad (39c)$$

$$v'_{1j} = v_{1j} - 2\mu\left(-E_1\frac{\partial S_t}{\partial v_{1j}}\right) \quad (39d)$$

$$v'_{2j} = v_{2j} - 2\mu\left(-E_2\frac{\partial T_t}{\partial v_{2j}}\right) \quad (39e)$$

Box 11

Note again the difference in sign as mentioned before. Also, the “sum” command in the code is due to the use of MSE form of Eqn. (21) (instead of Eqn. (20) which we used during derivation). This has been mentioned before.

Congratulations! We have completed our discussion, derivation and coding of the single-hidden layer neural network, which is key to our understanding towards Machine and Deep Learning.

The fully compiled code is given in the appendix.

References:

[1] Zhuhadar, Lily Popova & Lytras, Miltiadis. (2023). The Application of AutoML Techniques in Diabetes Diagnosis: Current Approaches, Performance, and Future Directions. Sustainability. 15. 13484. 10.3390/su151813484

[2] Sanubari, A. R., Kusuma, P. D., Setianingsih, C., Flood Modelling and Prediction Using Artificial Neural Network, The 2018 IEEE International Conference on Internet of Things and Intelligence System (IoTaIS)

[3] Michael Nielsen, Neural Networks and Deep Learning, http://neuralnetworksanddeeplearning.com

APPENDIX

```
clear, clc, close all

c1=3; c2=2; d1=2; d2=6;

% STEP 1 : Generate inital Value for weight and biases
M = 2;
v = rand(2,M)-1/2;
u = rand(1,M)-1/2;
w = rand(2,M)-1/2;

ER1 = []; %record error result
for iteration = 1:1000000

    % STEP 2 : Generate input [x,y]
    N = 1;
    L = 1;
    xk = (round(rand(1,N)*L,3));
    yk = (round(rand(1,N)*L,3));

    % STEP 3 : Calculate sigmoid function
    z = [];
    for j = 1:M %loop over neuron
        zj = w(1,j)*xk + w(2,j)*yk + u(j);
        z = cat(1,z,zj);
    end
    sig = 1 ./ (1 + exp(-z));
    dsigdz = sig .* (1-sig);

    % STEP 4 : Calculate neural network function
    St = 0;
    Tt = 0;
    for j = 1:M %loop over neuron
        St = St + v(1,j) *sig(j,:);
        Tt = Tt + v(2,j) *sig(j,:);
    end

    % STEP 5 : Calculate the actual equations
    s = c1*xk + c2*yk;
    t = d1*xk + d2*yk;

    % STEP 6 : Calculate the residual errors
    E1 = (s - St);
    E2 = (t - Tt);

    % STEP 7 : Calculate the gradients
    eta = 1e-2; %learning rate
    for j = 1:M %loop over neuron

        % diff over v1
        dStdv1 = -sig(j,:);

        % diff over v2
        dTtdv2 = -sig(j,:);

        % diff over u
        dStdu = -v(1,j) *dsigdz(j,:);
        dTtdu = -v(2,j) *dsigdz(j,:);

        % diff over w1
        dStdw1 = -v(1,j) *xk .*dsigdz(j,:);
        dTtdw1 = -v(2,j) *xk .*dsigdz(j,:);
```

```
        % diff over w2
        dStdw2 = -v(1,j) *yk .*dsigdz(j,:);
        dTtdw2 = -v(2,j) *yk .*dsigdz(j,:);

        % STEP 8 : Update the weight and biases
        v(1,j) = v(1,j) - eta*sum(2*E1.*dStdv1);
        v(2,j) = v(2,j) - eta*sum(2*E2.*dTtdv2);
        u(j)   = u(j)   - eta*sum(2*E1.*dStdu  + 2*E2.*dTtdu );
        w(1,j) = w(1,j) - eta*sum(2*E1.*dStdw1 + 2*E2.*dTtdw1);
        w(2,j) = w(2,j) - eta*sum(2*E1.*dStdw2 + 2*E2.*dTtdw2);

    end

    clc
    fprintf('No.of iteration = %d\n\n',iteration)
    disp('Matrix v1 v2 u w1 w2:')
    disp([v;u;w])
    fprintf('    s \t  St \t   E1  |   t      Tt      E2\n')
    fprintf('%6.2f %6.2f %8.4f |%6.2f %6.2f %8.4f\n',[s;St;E1;t;Tt;E2])

    Er1 = sqrt(sum(E1.^2 + E2.^2));
    ER1 = cat(1,ER1,Er1);
    fprintf('surd-G-squared error = %0.8f\n\n',Er1)

end
```

RESULT

```
No.of iteration = 500000

Matrix v1 v2 u w1 w2:
    4.4145    0.8866
    1.5626    6.9161
   -2.3067   -2.3421
    2.9438    0.6923
    1.3739    3.6857

    s      St      E1    |   t      Tt      E2
  3.05   3.07  -0.0139 |  2.56   2.53   0.0354
surd-G-squared error = 0.03806654
```