Introduction to Numerical Optimization

Assignment 3

May 2021

General Instructions:

- Submissions in pairs only.
- You are free to use either MATLAB or Python for any coding exercise.
- Sections labeled by [**Report**] should be included in your final report, and sections labeled with [**Code**] include a short coding task.
- Sections with no associated label include explanatory text only.
- More details about what should be included in your final report and your submission file are available in sections 2 and 3.
- **You are free to design your code however you want**.
- Due date - June 28th, 2021 (23:59).

1 Quasi-Newton Methods

In this section you will implement and experiment with the BFGS method, which is a Quasi-Newton optimization method. For your convenience, a good written introduction to the BFGS method and Quasi-Newton methods in general, can be found in chapter 6 of Numerical Optimization 2nd Edition, by Jorge Nocedal, Stephen Wright. Specifically, on page 140 you can find a good pseudo-code reference for the BFGS method.

1.1 BFGS Method Implementation [Code]

Implement a BFGS optimization routine which, given an objective function f and a starting point x_0, returns an approximation for a local minimum point of f. Please use the following implementation settings:

- **Stopping Criteria**: $||\nabla f(x_k)|| < 10^{-5}$
- **Inexact Line Search**: Backtracking with Armijo Rule (Step-size decrease factor $\beta = 0.5$, Slope decrease factor $\sigma = 0.25$, Initial step-size $\alpha_0 = 1$)
- **Initial Inverse Hessian Guess**: $H_0 = I$

Important Note (Updated: 27/05/2021): In order to ensure the positive-definiteness of H_{k+1} after the Hessian update step of the k-th iteration of the algorithm, you have to verify that the *Wolfe conditions* are satisfied. Specifically, update your line-search method such that it will satisfy the Wolfe conditions (page

33 on Numerical Optimization 2nd Edition), which are composed of the *sufficient decrease condition* and the *curvature condition*. Your Armijo rule implementation from Assignment 2 already satisfies the sufficient decrease condition, so you basically only have to add the curvature condition to your implementation.

1.2 Find the Minimum of the Rosenbrock Function [Code and Report]

Use the BFGS method to find the minimum point of the Rosenbrock function for the case of $n = 10$ and starting point $x_0 = (0, 0, \ldots, 0) \in \mathbb{R}^{10}$. As a reminder, the Rosenbrock function is given as follows:

$$f(x_1, x_2, \ldots, x_n) = \sum_{i=1}^{n-1} \left((1 - x_i)^2 + 100 \left(x_{i+1} - x_i^2 \right)^2 \right) \tag{1}$$

plot the convergence curve $f(x_k) - f^*$ (the y-axis) as function of the iteration number k (the x-axis), where f^* is the optimal/minimal value of the Rosenbrock function. Use logarithmic scale for the y-axis.

1.3 Neural Networks

In this section we will experiment with an example that demonstrate the universal approximation theory of feed-forward neural networks. The universal approximation theorems imply that neural networks can represent a wide variety of interesting functions when given appropriate weights. On the other hand, they typically do not provide a construction for the weights, but merely state that such a construction is

possible.

For those you who are not familiar with neural networks, even though we have briefly covered this subject in the context of computational graphs, I strongly recommend to watch the short introduction to this subject made by 3Blue1Brown.

In short, a feed-forward neural network is **simply** a non-linear function from $\mathbb{R}^n$ to $\mathbb{R}^m$ that is composed of modular layers - the output of the i-th layer is the input of the $(i+1)$-th layer. Feed-forward neural networks are known both theoretically and empirically for their extraordinary capabilities to approximate a wide variety of functions given enough data points sampled from the manifold we wish approximate/reconstruct. Each layer in the network possess a collection of parameters which can be tweaked in order to achieve the desired result. This "tweaking" process, which is also regarded in a fancy way as the *training process*, is articulated by an iterative optimization algorithm, which updates the network's parameters gradually at each iteration, until a predefined loss function, that is dependent on the network's output and the provided data points' ground-truth values (also called the *training dataset*), converges into a local minimum. Therefore, a feed-forward neural network is usually regarded as the *approximation model*, and the collection of the network's parameters are regarded as the *model parameters*.

In this exercise we will train a neural network to approximate the function given in equation 2 below, by using data points sampled randomly from its surface, as shown in figure 3.

$$f\left(x_1, x_2\right) = x_1 \cdot e^{-\left(x_1^2 + x_2^2\right)} \tag{2}$$

Where $(x_1, x_2) \in \mathbb{R}^2$ and hence $f : \mathbb{R}^2 \longrightarrow \mathbb{R}$. An example plot of the function in equation 2 is given in figure 2.

1.3.1 Neural Network Architecture

In order to approximate the function in equation 2, we will use the neural network architecture depicted in figure 1. Let us denote the neural network model by $F : \mathbb{R}^2 \longrightarrow \mathbb{R}$, such that given a set of model parameters $\mathcal{W}$ and an input data point $x \in \mathbb{R}^2$, the output of the network is denoted by $F\left(x|\mathcal{W}\right)$.

1.3.2 Activation Function

In this exercise, we will use the tanh activation function. That is, the non-linearity operation at the output of each neuron is given by:

$$\phi\left(x\right) = \tanh\left(x\right) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{3}$$

1.3.3 Loss Function (Updated: 27/05/2021)

In this exercise, we will use the mean squared error loss (MSE Loss), given as follows:

$$L\left(\left\{F\left(x^{i}|\mathcal{W}\right)\right\}_{i=1}^{n}, \left\{y^{i}\right\}_{i=1}^{n}\right) = \frac{1}{n}\sum_{i=1}^{n}\left(F\left(x^{i}|\mathcal{W}\right) - y^{i}\right)^{2} \tag{4}$$

Where $F\left(x^{i}|\mathcal{W}\right)$ and $\{y^{i}\}$ are the network output and ground-truth value of the i-th data point, respectively. In case you are a bit confused, please refer to the following visual illustrations:

- The training dataset $\{x^{i}, y^{i}\}_{i=1}^{n}$ is composed of n data points, as demonstrated in figure 3.

- The calculation of the loss for a single data point is depicted in figure 4.

Pay attention that the gradient of the loss function L with respect to the model parameters $\mathcal{W}$ is given by:

$$\nabla_{\mathcal{W}} L\left(\left\{F\left(x^i|\mathcal{W}\right)\right\}_{i=1}^n, \left\{y^i\right\}_{i=1}^n\right) = \frac{1}{n}\sum_{i=1}^{n} \nabla_{\mathcal{W}} \left(\underbrace{\left(F\left(x^i|\mathcal{W}\right) - y^i\right)^2}_{\text{Single Training Example Loss}} \right) \tag{5}$$

In other words, the gradient of the loss function with respect to the model parameters $\mathcal{W}$ is given by averaging the gradients of the loss of single training examples with respect to $\mathcal{W}$.

1.3.4 Explicit Expression of the Neural Network Model [Report]

Complete the following:

- Write an explicit vectorized expression for the neural network given in section 1.3.1. In other words, you should write the explicit expression of $F\left(x|\mathcal{W}\right)$. Please note, $\mathcal{W}$ represent the set of model parameters. That is, in our feed-froward neural network context, $\mathcal{W}$ consists of all weight matrices $\{W_i\}$ and all bias vectors $\{b_i\}$ across all layers. Even though your explicit expression for $F\left(x|\mathcal{W}\right)$ should be written in terms of x (the input vector), $\{W_i\}$, $\{b_i\}$ and the activation function ϕ, you can also think of $\mathcal{W}$, for implementation purposes, as a single long vector which is made of stacking the columns/rows of the matrices $\{W_i\}$ and the vectors $\{b_i\}$ one on top of the other. For guidance, please refer to figures 5 and 6.

- Specify the dimensions of each weight matrix W_i and bias vector b_i that you have used in your expression for $F(x|\mathcal{W})$. For example, if W_1 is a 2 by 2 matrix, you should specify that $W_2 \in \mathbb{R}^{2\times 2}$.

1.3.5 Evaluating f - The Function to be Approximated [Code] (Updated: 29/05/2021)

Write a routine that evaluates the function f from equation 2 on a single training example x^i given as input.

1.3.6 Evaluating the Activation Function's Value and Derivative [Code]

In this section we will treat the activation function as a standalone module. Implement the following routines:

1. Write a routine that calculates the value of the activation function given in equation 3 for a scalar input x.
2. Write a routine that calculates the first derivative of the activation function given in equation 3 for a scalar input x.

1.3.7 Evaluating the Loss Function's Derivative with Respect to Network's Output of Single Training Example [Code] (Updated: 29/05/2021)

In this section we will treat the loss function as a standalone module. Write a routine that calculates the derivative of the loss function with respect to the neural network's

output. That is, you should derive and implement $\frac{\partial L}{\partial F(x_i, \mathcal{W})}$.

1.3.8 Evaluate the Gradient of the Loss Function of a Single Training Example with Respect to $\mathcal{W}$ [Code] (Updated: 29/05/2021)

Write a routine receives a single input example $\{x^i, y^i\}$ and calculates gradient of the **loss function** given in equation 4 with respect to each of the neural network's model parameters (the weight matrices and bias vectors of the expression that you have deduced in section 1.3.4).

Guidance: This step can be easily implemented by following the steps below, assuming we're given an input example $\{x^i, y^i\}$:

1. Evaluate the forward-pass of the given input example through the neural network's layers, and store the output of each layer. That is, given the input example x^i and the set model parameters $\mathcal{W}$, you have to evaluate the output of the first layer, then use the output of the first layer as the input for the second layer, and so on - until you have evaluated the output of the last layer $F\left(x_i|\mathcal{W}\right)$.

2. Use the results from the previous bullet 1 and the input example's ground-truth value y^i to calculate $\frac{\partial L}{\partial F(x_i, \mathcal{W})}$ using your routine from section 1.3.7.

3. Use your results from bullet 2 to calculate the gradient of the loss function with respect to each of the inputs of the neural network's last layer. Use the expressions that were derived in slides 46-48 of the PowerPoint presentation about Computation Graphs.

4. Now that you have the gradient of the loss function with respect to the inputs of the last layer, you can back-propagate it into the previous layer, and repeat the process with respect to all remaining layers until you have calculated all the required gradients of the loss function. This repeated process is called the *back-propagation algorithm*, and in this exercise, we're implementing a tiny and dedicated version of it for our purposes. Please note, gradients have the same shape as the shape of their input. Therefore, a gradient with respect to a weights matrix $W \in \mathbb{R}^{m \times n}$ will also be a matrix in $\mathbb{R}^{m \times n}$. Hence, in order to reshape a gradient given in a matrix form in $\mathbb{R}^{m \times n}$ into a an equivalent gradient given in a vector form in $\mathbb{R}^{mn}$ and vice versa, we have to use a *reshape* operation. Please see numpy.reshape for Python and reshape for MATLAB. Moreover, during the optimization process (i.e. an optimization algorithm based on gradients, such as gradient descent or BFGS), we would like to stack together all the gradient vectors to form a single long gradient vector. Therefore, we have to use a *stack* operation. Please see numpy.concatenate for Python and vertcat for MATLAB.

1.3.9 Evaluate the Gradient of the Loss Function For the Entire Training Dataset with Respect to $\mathcal{W}$ [Code] (Updated: 27/05/2021)

Write a routine that receives a training dataset $\{x^i, y^i\}_{i=1}^n$ as an input, and calculates the gradient of the loss function with respect to the model parameters $\mathcal{W}$. Use your implementation of section 1.3.8 and equation 5.

1.3.10 Generate the Training Dataset [Code]

Write a routine that given n as an input, generates the training dataset $\{x^i, y^i\}_{i=1}^n$ by drawing n uniformly distributed random points $\{x^i\}_{i=1}^n$ from the 2D domain between $(-2, -2)$ to $(2, 2)$ and evaluating them on the function to be approximated using the routine you have written in section 1.3.5, in order to get the ground-truth values $\{y^i\}_{i=1}^n$. For drawing $\{x^i\}_{i=1}^n$, you might find the following methods helpful:

- numpy.random.rand in Python.
- rand in MATLAB.

1.3.11 Generate the Test Dataset [Code]

Write a routine that given n as an input, generates the test dataset by drawing n uniformly distributed random points $\{x^i\}_{i=1}^n$ from the 2D domain between $(-2, -2)$ to $(2, 2)$.

1.3.12 Model Parameters Initialization [Code]

Write a routine that initializes and returns the model parameters of the neural network you have deduced in 1.3.4 as follows:

- Each bias vector should be initialized with zeros.
- Each weights matrix $W \in \mathbb{R}^{m \times n}$ should be initialized such that each entry is drawn from the standard normal distribution and divided by $\sqrt{n}$. Please see

numpy.random.randn for Python and randn for MATLAB.

1.3.13 Data Visualization [Code]

Write a routine that given a function reference f, a collection of evaluated data points $\{x^i, y^i\}_{i=1}^n$ and a boolean b, plots a figure with the following elements:

- A surface plot of the function reference f evaluated over a mesh-grid between the points $(-2, -2)$ and $(2, 2)$ with step size 0.2. Please see the following implementation examples for Python and MATLAB.
- If b is true, draw a 3D scatter plot of $\{x^i, y^i\}_{i=1}^n$ on top of the surface plot. Please see the following implementation examples for Python and MATLAB.

1.3.14 Combining it All Together [Code and Report] (Updated: 27/05/2021)

In order to train the neural network and plot the results, follow the steps bellow:

1. Generate the training set for $n = 500$ using your implementation of section 1.3.10.
2. Generate the test set for $n = 200$ using your implementation of section 1.3.11.
3. Generate the initial model parameters $\mathcal{W}_0$ using your implementation for the initialization routine of section 1.3.12. Make sure $\mathcal{W}_0$ is reshaped as a long vector composed of the stacked weight matrices and bias vectors, as explained at the guidance paragraph of section 1.3.8.

4. For each $\epsilon \in \{10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}\}$:

   (a) Use your implementation of section 1.3.9 and the training dataset generated in step 1 to find a local minimum of the loss function of section 1.3.3 using your BFGS method implementation from section 1.1. Please use the following implementation settings:

   - **Stopping Criteria**: $||\nabla f(x_k)|| < \epsilon$
   - **Inexact Line Search**: Backtracking with Armijo Rule (Step-size decrease factor $\beta = 0.5$, Slope decrease factor $\sigma = 0.25$, Initial step-size $\alpha_0 = 1$)
   - **Initial Inverse Hessian Guess**: $H_0 = I$
   - **Starting Point**: $x_0 = \mathcal{W}_0$

   (b) Using the local minimum point $\mathcal{W}^*$ you have found in step 4a, plot the following two figures:

   - Draw a surface plot of $F(x|\mathcal{W}^*)$ using your implementation of section 1.3.13.
   - Evaluate your test dataset on $F(x|\mathcal{W}^*)$, and scatter plot it **on top** of a surface plot of equation 2 using your implementation of section 1.3.13.

   Please note, you might need to unpack $\mathcal{W}^*$ back into separate components representing the weight matrices and bias vectors.

Please explain the results you see in your plots.

2 What Should Be Included In Your Written Report? (Updated: 27/05/2021)

1. All of the explanations, plots and expressions/derivations that you were asked to give in sections 1.2, 1.3.4, 1.3.7, 1.3.14.

3 Assignment Submission

1. Please name your report's PDF file as **hw3_report_ID1_ID2.pdf**, and submit it through Coursera. **Only ONE individual of each pair has to submit this file**.

2. Please zip the source code files that you have written in all the coding sections into a file named **hw3_code_ID1_ID2.zip**, and submit your file through Coursera. **Only ONE individual of each pair has to submit this file**.

3. Please make a video recording with oral explanations to each of the following items:

   - Documentation of the code that you have written in section 1.1.
   - Documentation of the code that you have written in section 1.3.8.
   - Documentation of the code that you have written in section 1.3.14.

   You are free to record each video on your own pace and style.
   Please zip all your videos into a file named **hw3_videos_ID.zip**, and upload it through Coursera. **Please note, EACH individual has to record his**

own set of videos, and upload them SEPARATELY to Coursera. In case you experience problems with uploading the videos to Coursera, you can alternatively upload them to YouTube as unlisted videos, and submit instead a text file named **hw3_videos_ID.txt** with links to your YouTube videos. You are free to remove the videos from YouTube by the end of the semester.

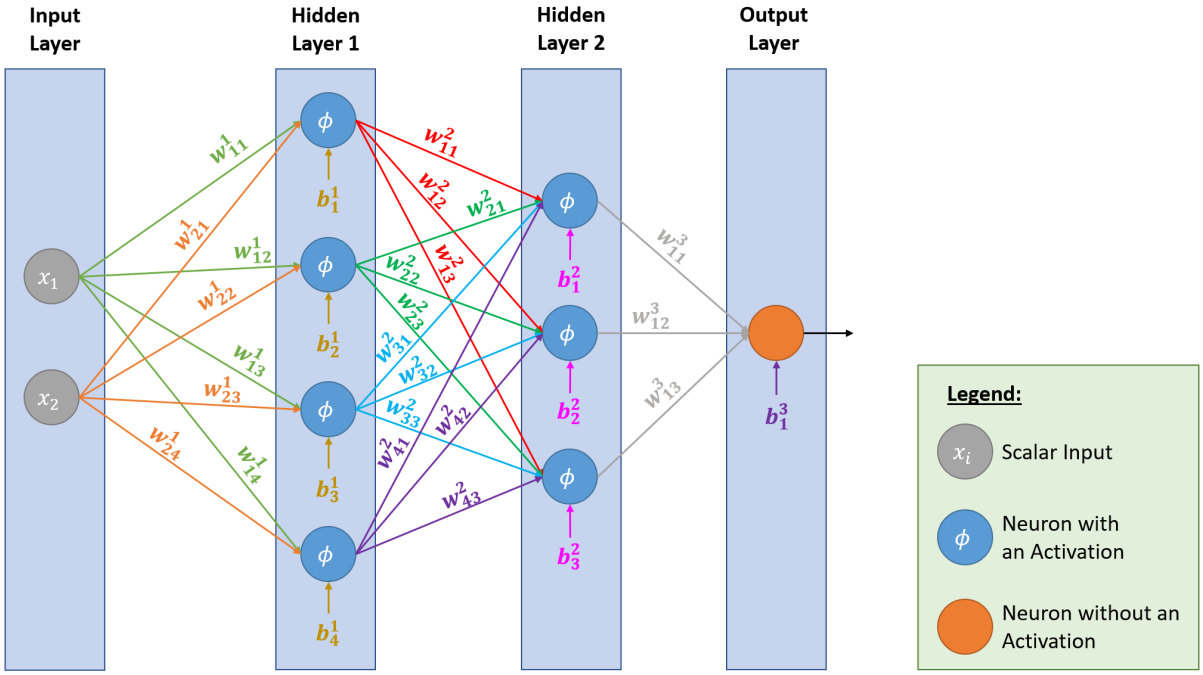

Figure 1: In this exercise we will implement a simple neural network with 4 layers. The first layer (the input layer) consists of two scalar inputs which forms a vector in $\mathbb{R}^2$. The next layer is the first hidden layer, which consists of 4 neurons, each with an activation function ϕ. Therefore, the output of the first hidden layer is a vector in $\mathbb{R}^4$. Next, we get to the second hidden layer, which consists of 3 neurons, each with the same activation function ϕ. Therefore, the output of the second hidden layer is a vector in $\mathbb{R}^3$. Lastly, we get to the output layer, which include a single neuron with no activation function applied on its output. Therefore, the output of the last layer is a scalar in $\mathbb{R}$. Please note, the colors of the edges, weights (w_{ij}^k) and biases (b_j^k) in the figure above have no meaning, it is just for making it easier to distinguish between them.

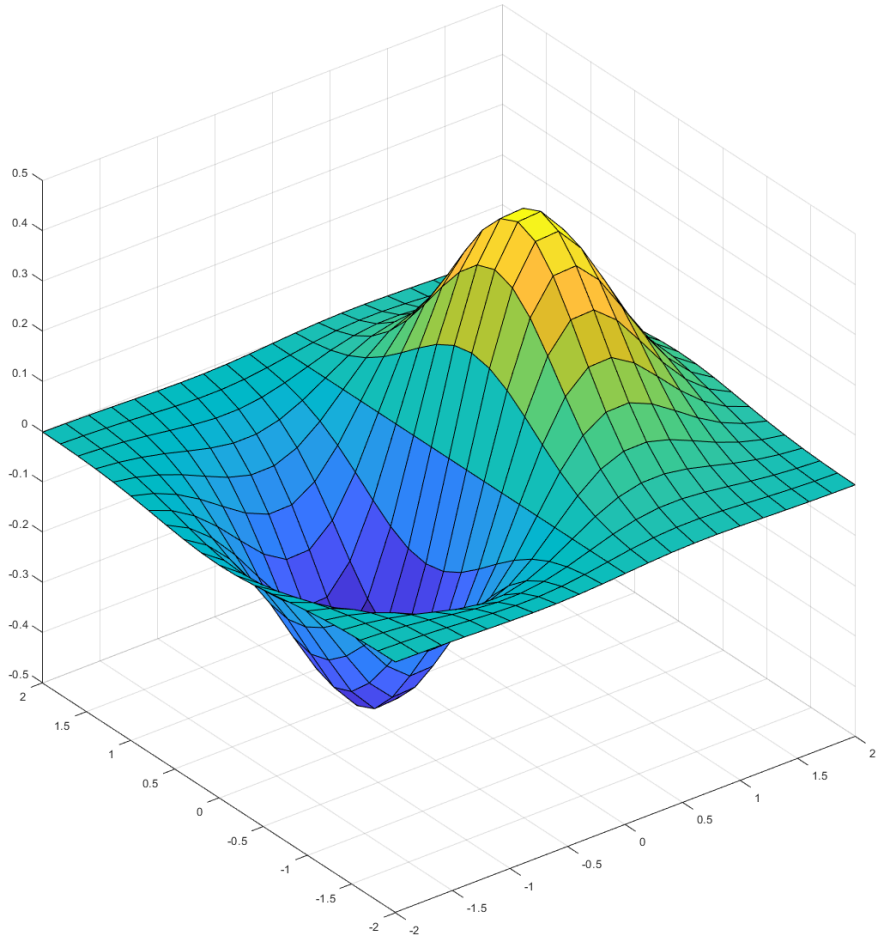

Figure 2: The plot of $f(x_1, x_2) = x_1 \cdot e^{-\left(x_1^2 + x_2^2\right)}$ over the 2D patch between $(-2, -2)$ to $(2, 2)$.

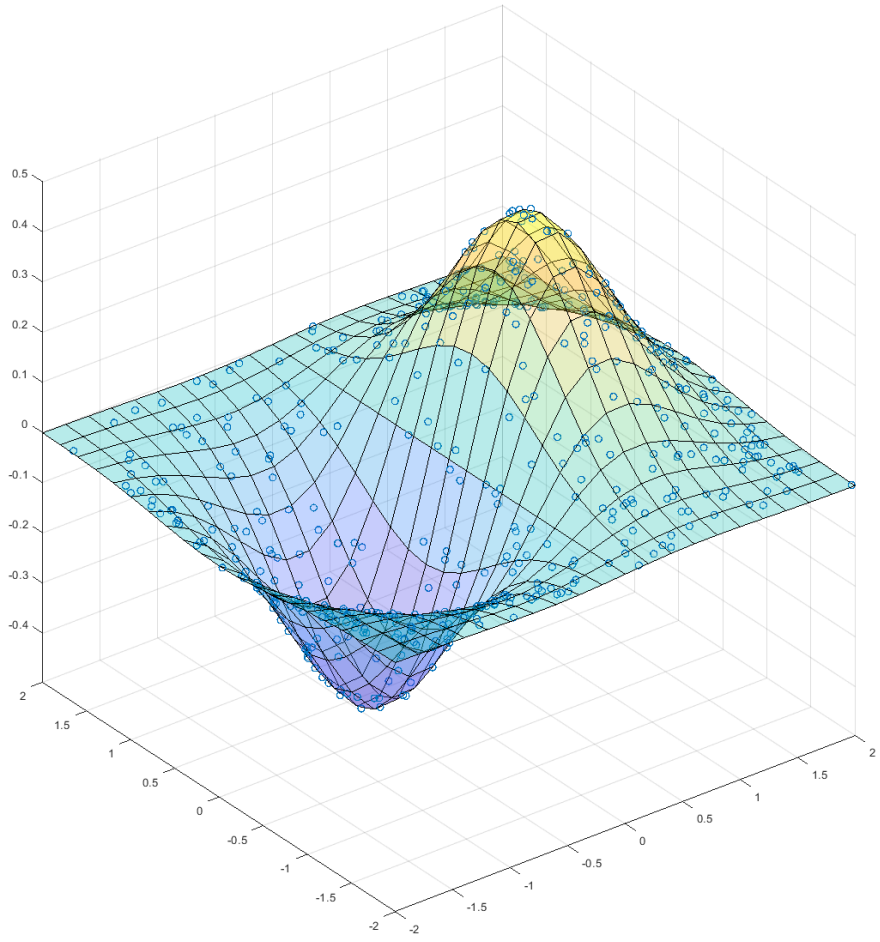

Figure 3: A collection of data points sampled from the surface of $f(x_1, x_2) = x_1 \cdot e^{-\left(x_1^2+x_2^2\right)}$ over the 2D patch between $(-2,-2)$ to $(2,2)$. These data points are an example for a training dataset which can be used to reconstruct $f(x_1, x_2)$ using a feed-forward neural network. Specifically, in order to generate such a dataset, we draw n random data points $\{x^i\}_{i=1}^n$ and evaluate them against the function f, which yields $\{x^i, y^i\}_{i=1}^n$ such that $f(x^i) = y^i$, where $x^i \in \mathbb{R}^2$ and $y^i \in \mathbb{R}$. **Please note, in "real life" scenarios, the function f will NOT be given to us, since if it was, the whole learning process was pointless. Instead, in such scenarios, the input training dataset will be provided by data acquisition techniques, and hopefully, the training process of the neural network will recover a good approximation to the original manifold from which the data points were collected.**

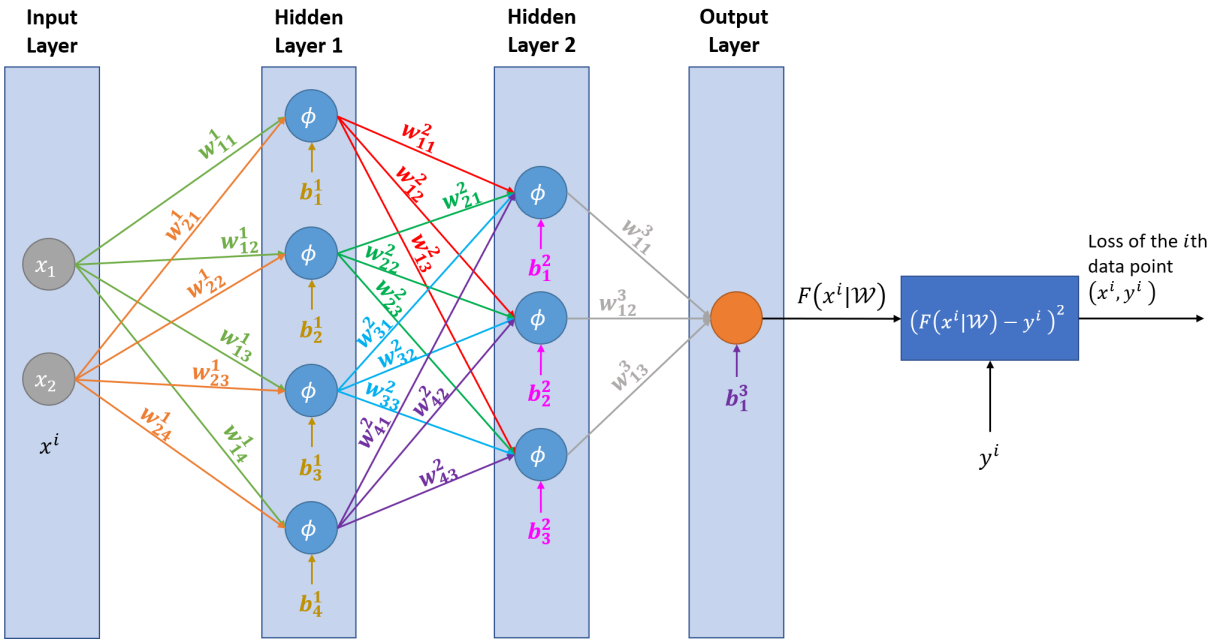

Figure 4: Loss calculation for an input data point (x^i, y^i).

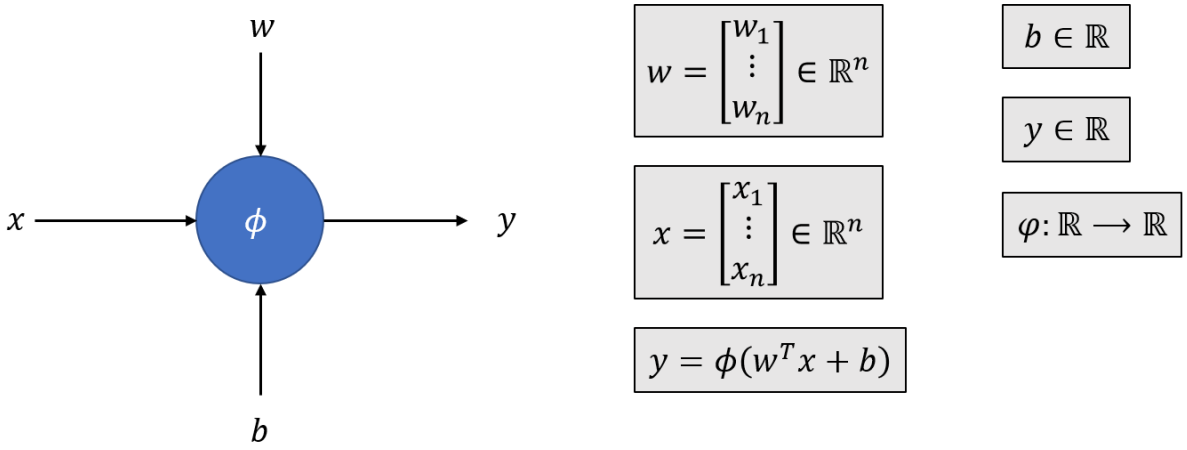

Figure 5: Vectorized expression of a single neuron.

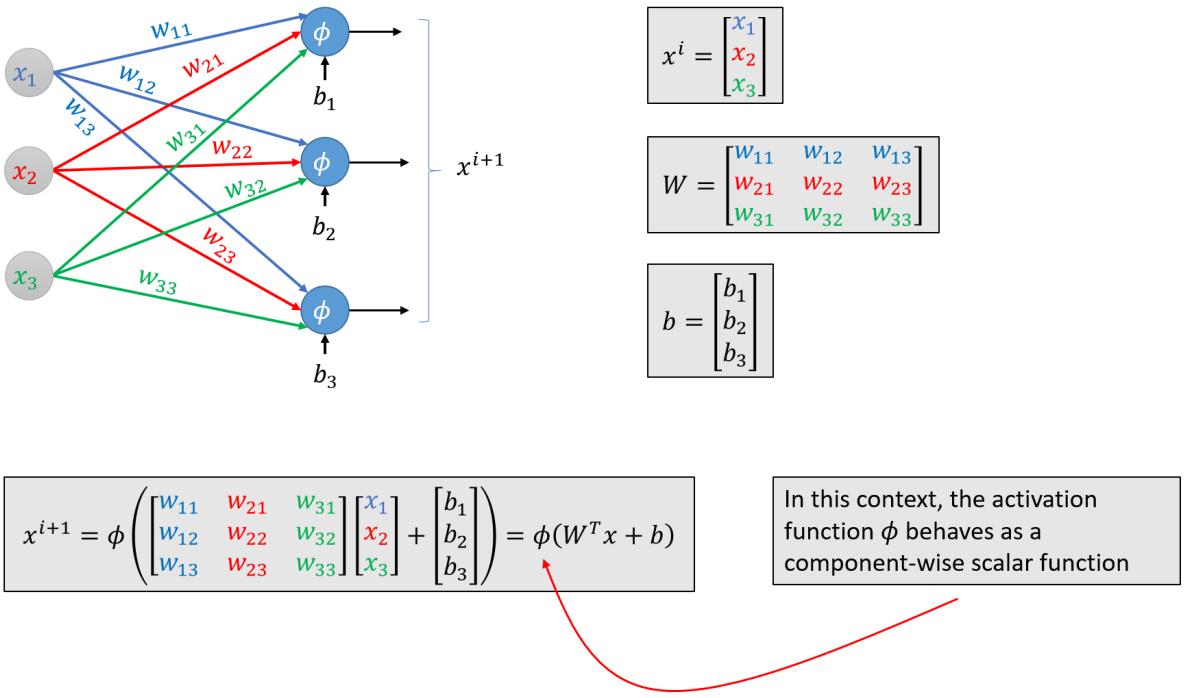

Figure 6: Vectorized expression of a complete neural network layer with 3 inputs and 3 outputs.