Deep Learning Assignment 1

Kok Teng, Ng (1936360), nkokteng

MinJeong, Lee (1978925), minjlee

Question 1: Implement an MLP

a. Examine the provided class Logistic Regression

i. Based on the provided $Logistic\ Regression\ class$, it was constructed by $an\ input\ layer$ and one hidden layer (with $log\_softmax(\ )\ function$).

There are having two parameters for initializing the *Logistic Regression* as ***D*** *and* ***C***, **D** is the number of inputs and **C** is the classes (as the width or number of neurons for the hidden layer)

As we known, the input layer will handle the input data, represented as a vector $x \in \mathbb{R}^D$ (or a matrix $x \in \mathbb{R}^{H*D}$ for multiple observations, with each row as an observation, columns as those variables, and ***H*** observations total)

The hidden layer will have two parameters as the matrix of ***weight*** and the vector of ***bias*** for the $Logistic\ Regression\ class$ before inserting those data into the activation function (as the $log\_softmax(\ )\ function$).

Weight W matrix denoted by $\boldsymbol{W} \in \mathbb{R}^{D*C}$. Regarding to the ***W*** matrix, the value of ***C*** is the current hidden state' width (as number of neurons, the numbers of output classes in the current setup of the $Logistic\ Regression\ model$) and the value of ***D*** will depend on the previous hidden layer width (which is the input layer's width or number of neurons).

Additionally, the ***bias b*** vector will have the dimensional as $b \in \mathbb{R}^C$, which having the same dimensional with the hidden layer.

As those elements within the ***W*** matrix and ***b*** vector is using the $torch.randn(\ )$ to initialize the value to them randomly. Then will use the $register\_parameter(\ )\ function$ to assign the name to each hidden layer's parameters as weight and bias, which can enable to access in the future. For instance, the provided code assigned the name to those hidden layers' parameters as "0_$weight$" and "0_$bias$", which are representing the weight and bias for those edges between the input and hidden layers.

ii. In the forward propagation of provided code, the calculation involves the ***dot product (matrix multiplication)*** of two matrices of weight matrix and the input data.

Then the code will transpose the weight matrix for getting the dimensional as $\boldsymbol{W}^T \in \mathbb{R}^{C*D}$, which is ready to do the dot product with the input data x

$$eta = self.W.t(\square)@x + self.b$$

we will get the dimensional of output after the dot product as $\mathbb{R}^C$ for the current hidden state, it will use the output to do the ***matrix additional*** with the pre-defined bias for the current hidden state with the dimension as $\boldsymbol{b} \in \mathbb{R}^C$. At the end, the output of this code "*eta*" has the dimensional of $eta \in \mathbb{R}^C$.

$$logprob = F.log\_softmax(eta, dim = -1)$$

It utilized the $log\_softmax(\ )$ function from the package of $torch.nn.functional$, the function will do the $softmax$ function as Element-wise application to eta vector as the following formula.

$$softmax(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

after applied the $softmax\ function$, we will get the probability of each element and the total of each row of those elements are ***1***. Then we will fit the $natural\ logarithm\ (ln)$ to those generated probability of each element from the vector as ***Element-wise application***

$$natural\ logarithm\ (ln)\ =\ e^{z_i}$$

generating the $logarithm\ probabilities\ value$ to each of element within the vector (or matrix). Additionally, the parameter of $dim\ =\ -1$ means it could handle the multi-dimensional tensors problem as the operation will be applied along the columns. At the end, the output dimension of $logprob\ \in \mathbb{R}^C$.

iii. The way to initialize and use the $Logistic\ Regression$ class, would require executing the following code as

$$logreg = LogisticRegression(3,2)$$

the first parameter is regarding as ***D*** (the input layer width as the number of inputs) and the second parameter is regarding as ***C*** (the hidden layer width as the generated number of output classes)

Based on the above code, the $Logistic\ Regression\ model$ will generate 3 neurons of the input layer and 2 neurons of for the hidden state as generate two classes as the output. The model is being named as $logreg(\ )$, we just could fit the data into the $logreg(\ )$ model could use the model.

As the following code,

$$logreg(x)$$

will generate the logarithm probability output for the input data.

If we would like to generate like to generate the probability, then we could use the exponential function to all elements within the vector of the model.

b. The code relevant of the implementation of the class MLP (Multi Layers Perceptrons) is located within the submitted Jupyter Notebook's cell 8. We were utilizing the $torch.randn(\ )\ function$ to generate the random values with normal distribution to the parameter of $weight\ \boldsymbol{W}$ and $bias\ \boldsymbol{b}$ with the dimensional as

$$W \in \mathbb{R}^{sizes[i-1]*sizes[i]}\ and\ b \in \mathbb{R}^{sizes[i]}$$

Based on the explanation and definition of each element within the sizes list, the length will be the number of layers for the MLP and the element of the list will be the width of the layer. We aforementioned those edges from previous layer forward propagation to the current layer are having two parameters as weight and bias, the weight matrix rows are depending on the previous layer, therefore, the row and column dimension of the weight $\boldsymbol{W}$ matrix is being defined as $sizes[i\ -\ 1]$ and $sizes[i]$ as the width (the output classes) for the current layer. On the other hand, the bias $\boldsymbol{b}$ vector are having the same dimension with the current layer's width as $sizes\ [i]$. However, the generated matrix by using the $torch.randn(\ )\ function$ is just a tensor matrix without wrapping as a neural network layer that is learnable parameter during the training for tracking the gradient.

Then we were utilizing the $torch.nn.Parameter(\ )$ function to wrap up the matrix and generate the parameter for the according layer that could learnable during the training (as backward propagation) for tracking and computing the gradient via the PyTorch framework. We also utilized the

$register\_parameter(\,)$ function to assign the name for each parameter, such as those variables for those edges from input layer to the first hidden layer as "0_weight" and "0_bias", so the until the output layer it will be $f"\{len(sizes) - 1\}\_weight"$ and $f"\{len(sizes) - 1\}\_bias"$

During the $def\ forward(self, x)$, we were going to do the for loop to go through every layer for executing the forward propagation. As we will use the $getattr(\,)\ function$ to get the according layer weight and bias matrix and vector for doing the further feedforward process during every layer (as for looping). After getting those parameters, we were going to do the ***dot product*** for the weight matrix $\boldsymbol{W}$ and the input data $\boldsymbol{x}$ then the output of dot product will do the ***matrix additional*** with the bias vector. As the following code,

$$x = weight.t(\,) @ x + bias$$

since we know the dimension of the input data is $x \in \mathbb{R}^{sizes[i-1]}$ before the dot product then we are going to do the dot product with the transposed weight matrix with the dimension as $W \in \mathbb{R}^{sizes[i]*sizes[i-1]}$, then both of them could only perform the ***dot product (matrix multiplication)*** due to the constraints of the matrix. Therefore, the equation will be $W_i^T * x$. The generated output will be the dimension as $\mathbb{R}^{sizes[i]}$ and doing the matrix addition with the bias $b \in \mathbb{R}^{sizes[i]}$, as the final output (matrix additional output) will still with the dimension of $\mathbb{R}^{sizes[i]}$.

The matrix additional output will be fitted into the ***activation function*** (the default function is ***sigmoid*** or ***logit function***). The above computation will execute until the second last hidden layer, due to the ***output layer will only going to use the linear activation function*** to generate the output. If we are continue using the ***Rectified Linear Neuron (ReLU)*** at the output layer, it might have the potential as before fitting the output into the activation function is negative value then it will return 0 as the output that is not, we expected.

For defining the output layer is linear activation function, we were not including it in the for loop mechanism. We separated them and using the following code for getting the output layer's needed parameters such as weight and bias, as

$$weight = getattr(self, f"\{self.num\_layers(\,) - 1\}\_weight")$$

$$bias = getattr(self, f"\{self.num\_layers(\,) - 1\}\_bias")$$

that could help us to get the parameters for the output (last hidden state) layer and doing the linear activation to it without applying the default (sigmoid or logit function) or ReLU activation function based on the requirement of the question. The dot product between input data with the weight matrix and the matrix additional with bias are same as above mentioned for the output layer (as last hidden layer with linear function).

The reason of the output of each hidden state keeping in the ***variable x*** is the current hidden state's output will be the next hidden state's input, so we could define it in this way during the practical.

c. The code relevant to the expand of the implementation of the class MLP that programmed in **question 1.b** is located within the submitted Jupyter Notebook's cell 12. We were doing a small modification to the code for enabling to handle the vector and matrix as input data.

At the beginning of the code, we were using the if-else statement to check the dimension of the input data, if it is a vector then we are going to use the $unsqueeze(0)\ function$ to the input data as converting them into a two-dimensional matrix. For instance, the vector as $tensor([1, 2, 3])$, after the unsqueeze function, will be $tensor([[1, 2, 3]])$ that are enabled to do the forward pass without using the built-in $vmap()\ function$ from PyTorch package as the following code:

$$if\ x.dim == 1 : \{x = x.unsqueeze(0)\}$$

Each row of the input data will be represented as an observation to be fitted into the model for forward passing.

This also applied to bias b since bias is a vector, we must $unsqueeze(0)$ it then it will have the dimension of $\boldsymbol{b} \in \mathbb{R}^{1 * sizes[i]}$ that is enable to do the ***matrix additional*** for the bias with the ***dot product*** result matrix. The matrix additional will add the bias vector for each row of the dot product result matrix as Element-row application, by using the following code as:

$$x = x\ @\ weight + bias.unsqueeze(0)$$

The reason we have to use the equation of $x\ @\ weight$ instead of $weight.t()\ @\ x$ is the input data x is a matrix with the dimension of $x \in \mathbb{R}^{H * sizes[i-1]}$, H is the rows of observations and $sizes[i-1]$ is the variable of the data, then we have to use it to multiply with the weight with the dimension of $\mathbb{R}^{sizes[i-1] * sizes[i]}$. Therefore, we must convert the equation into $x\ @\ weight$ to handle the matrix constraints during executing the dot product of two matrices. Then the dot product output's dimension will be $\mathbb{R}^{H * sizes[i]}$ and able to do the matrix additional with the bias as $\boldsymbol{b} \in \mathbb{R}^{1 * sizes[i]}$ with Element-row application.

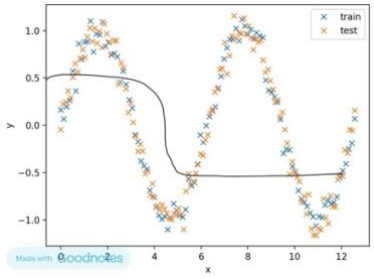
Then the remaining part of the forward propagation's concept is same with the code from **question1. b.**

Question 2: Experiment with MLPs

a. To conjecture how a fit of Feedforward Neural Network (FNN) with different number of neurons (with using the sigmoid as the activation function) will look like, as:

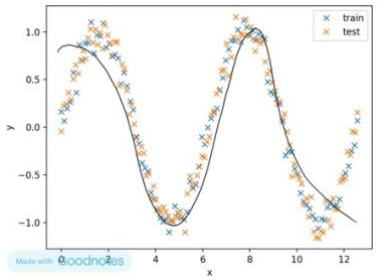
i. Zero-Hidden Neuron: When there are having zero neuron within the FNN model, it will be considered as a kind of linear model, and the fit of FNN model will be a straight line as a hyperplane in a higher dimension. The hyperplane could only handle the linear relationships between the input features and output. It will show a hyperplane within the graph as a linear straight line.

ii. One-Hidden Neuron: The FNN model with one hidden neuron (as width) could capture simple non-linear patterns, such as the fit of FNN model will approximate be a curve or simple transformation for capturing simple non-linear patterns between the input features and output. Based on the FNN setup with using the sigmoid function for the neuron, we could expect there might be a sigmoid curve shown within the graph, but it doesn't fit well to the training and testing dataset, as the illustration shown at the bottom.

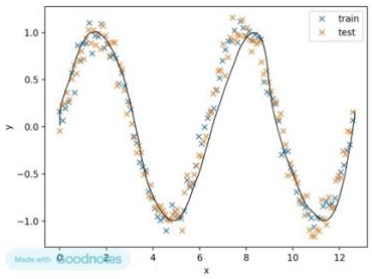


iii. Two-Hidden Neuron: When the FNN with two hidden neurons with sigmoid function, it could capture more complex non-linear mapping patterns such as a bend curve with 2 curves. Each hidden neuron introduces a curve, and together they form a more complex fit line. We could expect there might a fit line

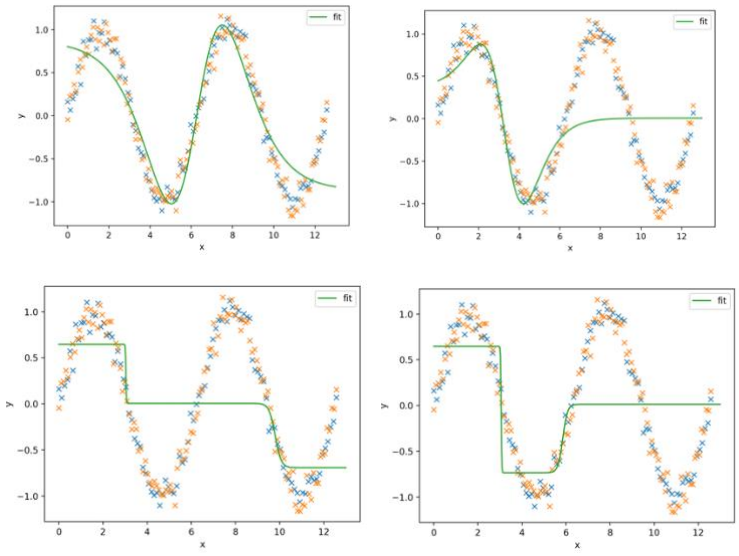
starting to perfectly fit with the training and testing dataset and approximate the underlying data distribution, as the illustration shown at the bottom.



iv. Three-Hidden Neuron: The FNN model could provide even greater flexibility on capturing those non-linear mapping pattern with more complex functions, such as multiple bends or transition with 3 curves. We could expect there might a fit line perfectly fit to the training and testing dataset since it has three hidden neurons (with sigmoid function) to handle those non-linearly problem, as the illustration shown at the bottom.



b. By using the provided code for training an FNN model with two hidden neurons based on the Scipy's BFGS optimizer, we could get several results when we execute the code (as training multiple times) several times but those result is what we expected to see as the fit of FNN could capture complex non-linearly input variables through the bend curve and provide more flexibility to the model, as these illustrations shown at the bottom.



Universität Mannheim, ng.kok.teng@students.uni-mannheim.de, Mannheim Master in Data Science (MMDS)

Universität Mannheim, minjlee@students.uni-mannheim.de, Mannheim Master in Data Science (MMDS)

After repeating (re-execute) training the FNN model with two hidden-neurons multiple times, we will get different results in every re-execution but will get one of the fit lines as the above illustrations.

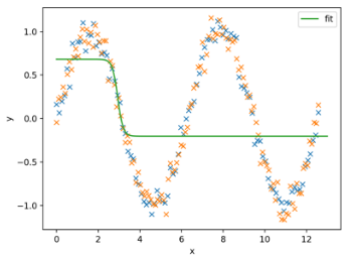
The reason that we will get different fit results from each re-execution (training) is we do not initialize the $torch.manual\_seed(\,)$ to ensure consistent weight and bias parameters to the model. Therefore, the model's weight and bias are initialized randomly in each (reps), which will resulting in different outputs for each neuron.

Based on the $def\ train1(\,)$ code, we could know that the $nreps$ parameter is being used to train the number of new models and returning us a model as the best model with generating the least error and largest gradient evaluations. It doesn't update the weight and bias values after everything training of computing the error by using ***Mean Squared Error (MSE)*** and BFGS as it just initializes a new model to train it again and use the ***MSE*** to compare the loss between each model as minimizes the error. The above actions don't update the weight and bias of the model, it just doing the comparison of each model with different initialized weight and bias value.

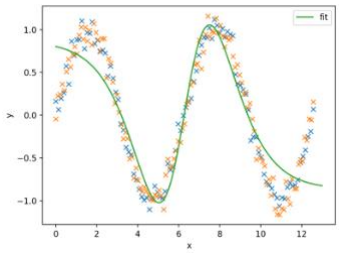
For instance, we could investigate to the cell 23 within the submitted ***nreps*** Notebook, we passed the value of 10 to the parameter of ***nreps***. It indicates as it will train the 10 models and returning us a single model as the best model with the least error and largest gradient evaluations.

The MSE function computes the model's error by comparing the predicted result with the actual result. As the error increases, the loss grows since we are using ***Newton's method*** to generate the gradient. This leads to larger gradients and consequently different errors for the model. This is a key challenge when investigating how the weights influence the loss and gradient. Interestingly, the model tends to generate a smaller error (MSE) when the gradient of the parameter is small. This suggests that the model with smaller gradient perform better and fit the training data more accurately. This reflected in the larger number of gradient evaluations, indicating a more thorough exploration of the parameter space.

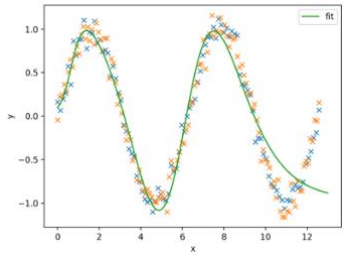
c. At the bottom, we have shown the illustration about the fit line of FNN model was being trained with different numbers of hidden neurons (such as 1, 2, 3, 10, 50, and 100).



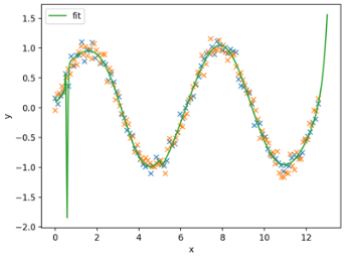
Trained with 1 Hidden Neuron



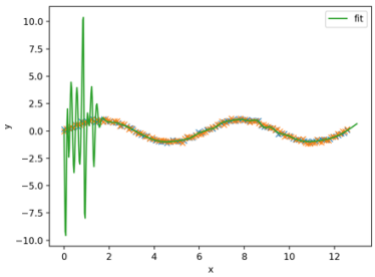
Train with 2 Hidden Neurons



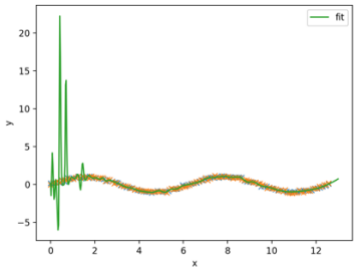
Trained with 3 Hidden Neurons



Trained with 10 Hidden Neurons



Trained with 50 Hidden Neurons



Trained with 100 Hidden Neurons

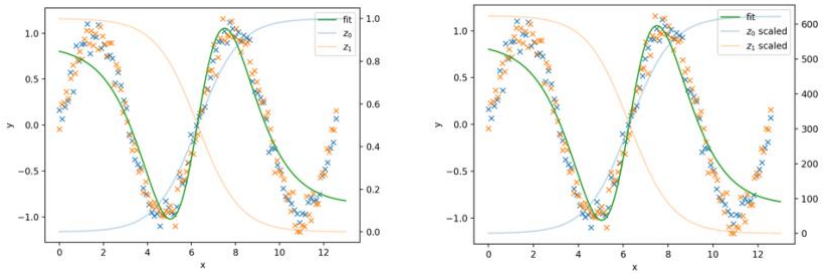
Based on the illustration shown above, we could observe that when the number of hidden neurons increase, the FNN fit line is getting more bend curves for handling the more complex non-linearly input variables and output. Those FNN fit is what we expected too since we mentioned that each neuron will introduce a curve, then they will form a more complex fit line together. As when there are having more hidden neurons, the bend curve will be more complex for capturing and handling more complex relationship.

Since each hidden neuron will provide their own distributed representation for capturing different features or patterns from the input data, when there are having more hidden neuron then it will have more distributed representation for the FNN model such as learning non-linear pattern data within a higher-dimensional space. At the end, the model will combine all distributed representations for generating a single fit line to the network can approximate complex functions, more distributed representation will lead to more bend curves to the neural network model (such as FNN).

However, there might lead to a problem as overfitting due to the model is too complex and fit the training data too well as only memorize the random noise in the training data, which doesn't understand and learn the underlying pattern and knowledge of the training data. We could observe from the FNN model with 10, 50, and 100 hidden neurons was being trained is overfitting, as it doesn't mean the more hidden neurons the model will fit greater to the input variables and output. Additionally, when the number of hidden neurons is too less, it will cause the model underfitting as doesn't fit the data nicely. For instance, when the MLP with a hidden neuron is underfitting, where the fit line is too simple to model and make it doesn't capture the non-linearly data nicely.

Therefore, we could use the Universal Approximation Theorem or Ablation as guarantee the existence of an FNN model that can represent any continuous function with enough hidden layers and number of hidden neurons (but finitely) to handle the task on the hand or remove or add some of the components (such as number of neurons or depth of the model) from the model at a time to test its impact with different network architectures on performance.

d. Those illustrations shown at the bottom as the FNN model with different numbers of hidden neurons (as 2, 3, and 10), on those left graphs represented the distributed representation or embedding of the input data and those right graphs represents the scaled by using the weight of their respective connection to the output of the hidden neurons.

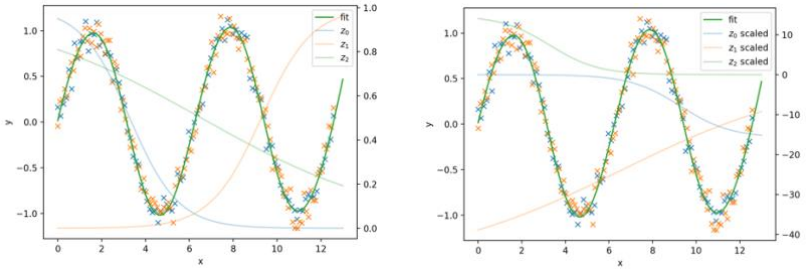


FNN model with 2 hidden neurons

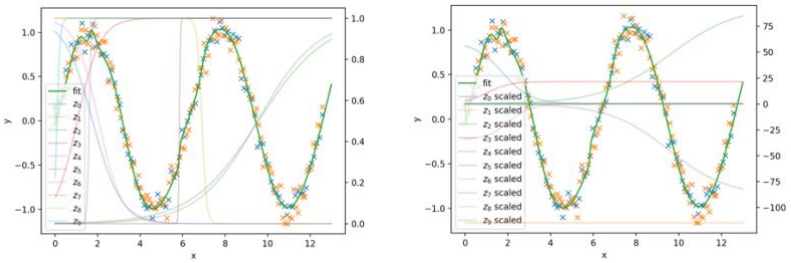
The above illustrations shown the FNN model trained with 2 hidden neurons using synthetic data, and the left graph represent the distributed representations of the input data that generated by each hidden neuron (as z_1 and z_2 within a higher-dimensional space. Each hidden neuron contributes to the overall representation for capturing different features or patterns and each neuron will focus on specific features, leading to a richer and more informative representation when the distributed representations are more expressive and allow the FNN to handle more relationships.

Each neuron contributes to shaping these boundaries, capturing different aspects of the data distribution. At the end, the model will combine all distributed representations' output to generate a fit line to the network (FNN model) can approximate complex functions, as handling more complex non-linear data and performing well. The finalized fit line of FNN model will be represented in a higher-dimensional space where pattern emerge, the network can then navigate this space to make predictions.

Since, we obtain the distributed representation we could use the connection weights to scaled the output of the hidden neurons, this could highlighting their contributions to the final output for showing how much each hidden neuron affects the overall prediction.



FNN model with 3 hidden neurons



FNN model with 10 hidden neurons

When we were increasing the number of hidden neurons as 3 and 10 hidden neurons, the FNN becomes more flexible and can capture more patterns in the data. Since the distributed representation of the model becomes richer due to each hidden neuron must generate their own embedding, which allowing the FNN to approximate intricate relationships.

The distributed representation is intuitive and flexibility as the FNN model's flexibility could let it to learn those non-linear transformations of input data. Each hidden neuron acts as feature extractors, transforming the input data into a higher-dimensional space for enabling to create the boundary decision to capture different aspects of the data to generalize well on those unseen data. However, when we are trying to be interpreting an individual hidden neuron may not be intuitive.

Question 3: Backpropagation

a. In the submitted Jupyter Notebook, you will find the forward pass code for the given experiment setup in cell 34. We have implemented essential operations from the question PDF, including matrix dot products (handling the dimension between matrices), matrix addition, activation functions applied to vector as Element-wise application, and computation of the loss function using $Squared\ Error\ (SE)$ between the predicted result $\hat{y}$ and the actual result y without using the built-in PyTorch function. Those generated prediction result of $\hat{y}$ and loss l are same as all the outputs provided by you and professor.

b. The code relevant to this question as backward propagation is located in cell 35 of the submitted Jupyter Notebook. In the backward propagation, it cleverly employs the chain rule and reuses previously computed derivatives (gradients) to replace with part of the partial derivative of the current gradient to optimize computational efficiency. At the bottom, it shown the derivations and resulting formulas for each of those required quantities.

$$\delta l = \frac{\partial l}{\partial l} = 1$$

$$\delta \hat{y} = \frac{\partial l}{\partial \hat{y}} = \frac{\partial (y - \hat{y})^2}{\partial \hat{y}} = 2 * (y - \hat{y}) * (-1) = -2 * (y - \hat{y})$$

$$\delta y = \frac{\partial l}{\partial y} = \frac{\partial (y - \hat{y})^2}{\partial y} = 2 * (y - \hat{y})$$

$$\delta b_2 = \frac{\partial l}{\partial b_2} = \frac{\partial l}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial b_2} = \delta \hat{y} * \frac{\partial (z_4 + b_2)}{\partial b_2} = \delta \hat{y} * 1$$

$$\delta z_4 = \frac{\partial l}{\partial z_4} = \frac{\partial l}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z_4} = \delta \hat{y} * \frac{\partial (z_4 + b_2)}{\partial z_4} = \delta \hat{y} * 1$$

$$\delta w_2 = \frac{\partial l}{\partial w_2} = \frac{\partial l}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z_4} * \frac{\partial z_4}{\partial w_2} = \delta z_4 * \frac{\partial (w_2^T * z_3)}{\partial w_2} = \delta z_4 * z_3$$

$$\delta z_3 = \frac{\partial l}{\partial z_3} = \frac{\partial l}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z_4} * \frac{\partial z_4}{\partial z_3} = \delta z_4 * \frac{\partial (w_2^T * z_3)}{\partial z_3} = \delta z_4 * w_2^T$$

$$\delta z_2 = \frac{\partial l}{\partial z_2} = \frac{\partial l}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z_4} * \frac{\partial z_4}{\partial z_3} * \frac{\partial z_3}{\partial z_2} = \delta z_3 * \frac{\partial \sigma(z_2)}{\partial z_2} = \delta z_3 * \sigma(z_2) * [1 - \sigma(z_2)]$$
$$= \delta z_3 * z_3 * (1 - z_3)$$

$$\delta b_1 = \frac{\partial l}{\partial b_1} = \frac{\partial l}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z_4} * \frac{\partial z_4}{\partial z_3} * \frac{\partial z_3}{\partial z_2} * \frac{\partial z_2}{\partial b_1} = \delta z_2 * \frac{\partial (z_1 + b_1)}{\partial b_1} = \delta z_2 * 1$$

$$\delta z_1 = \frac{\partial l}{\partial z_1} = \frac{\partial l}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z_4} * \frac{\partial z_4}{\partial z_3} * \frac{\partial z_3}{\partial z_2} * \frac{\partial z_2}{\partial z_1} = \delta z_2 * \frac{\partial (z_1 + b_1)}{\partial z_1} = \delta z_2 * 1$$

$$\delta x = \frac{\partial l}{\partial x} = \frac{\partial l}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z_4} * \frac{\partial z_4}{\partial z_3} * \frac{\partial z_3}{\partial z_2} * \frac{\partial z_2}{\partial z_1} * \frac{\partial z_1}{\partial x} = \delta z_1 * \frac{\partial (w_1^T * x)}{\partial x} = \delta z_1 * w_1^T$$

$$\delta w_1 = \frac{\partial l}{\partial w_1} = \frac{\partial l}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z_4} * \frac{\partial z_4}{\partial z_3} * \frac{\partial z_3}{\partial z_2} * \frac{\partial z_2}{\partial z_1} * \frac{\partial z_1}{\partial w_1} = \delta z_1 * \frac{\partial (w_1^T * x)}{\partial w_1} = \delta z_1 * x$$