RDFIA : Pattern Recognition and Machine Learning for Image Understanding

Deep Learning Practical Work 1-a, 1-b

Introduction to neural networks

Srijita NANDI, Seydina KANDE
M2 Quantum Information

1 Introduction

In this lab, we explore fundamental concepts in neural networks, focusing on building a simple network from scratch to understand core principles such as forward and backward propagation, parameter initialization, and optimization. We implement functions for forward passes, calculate losses and accuracy, and apply gradient descent for model updates. By simulating these essential neural network operations manually, we gain a deeper understanding of how data flows through a network, how adjustments are made through backpropagation, and how these steps contribute to model learning.

2 Theoretical foundation

2.1 Supervised dataset

Questions:

1. What are the train, val and test sets used for?
   Answer: **Train Set:** The train set helps our model to make a guess, and measure how well it did and then repeat but next time, make a reasonable guess. This is what we call learning phase in which model learns the hidden pattern and insight from the data that is helpful for the task we want to do, be it a regression, classification, ranking, etc.
   Validation Set: The validation set is used to evaluate and fine-tune a machine learning model during training, helping to assess the model's performance and make adjustments. By evaluating a trained model on the validation set, we gain insights into its ability to generalize to unseen data. This assessment helps identify potential issues such as overfitting, which can have a significant impact on the model's performance in real-world scenarios.
   Test Set: The train set serves as an unbiased measure of how well the model generalizes to unseen data, assessing its generalization capabilities in real-world scenarios. By keeping the test set separate throughout the development process, we obtain a reliable benchmark of the model's performance. The test dataset also helps gauge the trained model's ability to handle new data. Since it represents unseen data that the model has never encountered before, evaluating the model fit on the test set provides an unbiased metric into its practical applicability. This assessment enables us to determine if the trained model has successfully learned relevant patterns and can make accurate predictions beyond the training and validation contexts.

2. What is the influence of the number of examples N ?
   Answer: In a supervised classification task, the number of examples N in the dataset plays a critical role in several aspects of model performance and generalization.
   A larger dataset provides the model with more diverse examples, helping it learn more generalizable patterns rather than overfitting to specific training data. With more training data, models tend to improve in accuracy and generalization to unseen data (test/validation sets), reducing the variance of the model. However, the benefit of increasing N diminishes after a certain point, depending on the complexity of the task and the model capacity.
   With fewer examples, the model might struggle to capture the underlying distribution of the data, leading to overfitting or underfitting. It may perform well on training data but poorly on the test set.

2.2 Network architcture (forward)

Questions:

3. Why is it important to add activation functions between linear transformations?
   Answer:A linear transformation maps inputs to outputs by applying a linear function (e.g., matrix multiplication). Stacking multiple linear transformations results in another linear function. This means that no matter how many layers you add, the entire network would essentially be equivalent to a single-layer linear model.By introducing a non-linear activation function between linear layers, the network can model complex, non-linear relationships. This allows the network to approximate any arbitrary function, making it possible to solve problems that involve complex patterns in data, such as image recognition, natural language processing, and other tasks in deep learning as well. Popular activation functions (e.g., ReLU, sigmoid, and tanh) are chosen for their properties of being smooth and differentiable, which helps the optimization process.

4. What are the sizes $n_x; n_h; n_y$ in the figure 1? In practice, how are these sizes chosen?
   Answer:
   n_x represents the number of features in the input vector x . It corresponds to the dimensionality of the input data. This size is determined by the nature of the input data. For example, in an image classification task, if the input is a grayscale image of size 28×28 pixels, then $n_x = 28 \times 28 = 784$. For tabular data, each feature (or column) in the dataset is treated as an input dimension. So, if there are 10 features, $n_x = 10$.
   n_h represents the number of units (or neurons) in the hidden layer. It determines the number of intermediate representations or features learned by the network. This size is hyperparameter-dependent. It is usually chosen based on the complexity of the task and the amount of data available. In order to choose the n_h one may start with a size comparable to or larger than n_x and tune it through cross-validation, like trying powers of 2 (e.g., 64, 128, 256) or other ranges depending on the problem's complexity.
   n_y represents the number of output classes in the classification task or the number of outputs in a regression task.In a classification task, n_y is equal to the number of classes. In regression, it equals the number of outputs being predicted.

5. What do the vectors $\hat{y}$ and y represent? What is the difference between these two quantities?
   Answer: $\hat{y}$ is the predicted output, the output vector produced by the neural network after processing the input through its layers and applying the final activation function (e.g., SoftMax in classification tasks). It represents the network's predictions.
   y is the actual, true value or label associated with the input data. It is provided in the supervised dataset during training. It represents the ground truth the model is trying to learn to predict.
   Differences:

   - During training, the model's goal is to make $\hat{y}$ as close as possible to y by adjusting its parameters (weights and biases). The difference between $\hat{y}$ and y (often measured by a loss function like cross-entropy for classification or mean squared error for regression) drives the backpropagation process, which updates the model's weights to reduce prediction errors.
   - In classification, $\hat{y}$ contains predicted probabilities, while y is a binary one-hot encoded vector indicating the actual class. In regression, $\hat{y}$ is a continuous value (or vector), which ideally should match the continuous true value y .
   - Example: For a classification task with three classes:
     $\hat{y} = [0.2, 0.5, 0.3]$: The neural network predicts class 2 with 50% probability, class 1 with 30%, and class 0 with 20%.
     $y = [0, 1, 0]$: The true label is class 1, represented by the one-hot encoded vector.

6. Why use a SoftMax function as the output activation function?
   Answer: The SoftMax function is used because we are working with a multi-class classification task here and SoftMax is commonly used as the output activation function in neural networks for multi-class classification tasks.
   Raw outputs (logits) from the final layer of the network are often unbounded real numbers and don't have any probabilistic interpretation. The SoftMax function converts these logits into probabilities, where each value lies between 0 and 1, and the sum of all probabilities equals 1. This is important because, in classification tasks, we often want to interpret the output of the network as the probability of each class. Also, SoftMax is differentiable, which is essential for training neural networks using gradient-based optimization methods like backpropagation. The gradients of the loss function with respect to the model's parameters can be computed and used to update the weights. In comparison to other activation functions like the sigmoid, which produces independent outputs for each class (not summing to 1), SoftMax ensures that the probabilities of all classes are linked (sums up to 1), such that increasing the probability for one class decreases the probabilities of others.

7. Write the mathematical equations allowing to perform the forward pass of the neural network, i.e. allowing to successively produce $\tilde{h}, h, \tilde{y}$ and $\hat{y}$ starting at x.
   Answer:
   (a) Computing the pre-activation of the hidden layer ($\tilde{h}$) which involves an affine transformation of the input vector x using the weight matrix W_h and the bias vector b_h for the hidden layer.
   $\tilde{h} = W_h x + b_h$
   (b) Applying the activation function to the hidden layer (h):
   $h = \tanh(\tilde{h})$
   $=> h = \tanh(W_h x + b_h)$
   (c) Computing the pre-activation of the output layer ($\tilde{y}$):
   $\tilde{y} = W_y h + b_y$
   (d) Applying the activation function to the output layer ($\hat{y}$):
   $\hat{y}_i = \frac{e^{\tilde{y}i}}{\sum j e^{\tilde{y}_j}}$ for each i

2.3 Loss function

8. During training, we try to minimize the loss function. For cross entropy and squared error, how must the $\hat{y}_i$ vary to decrease the global loss function L?
   Answer: For a single example, the cross-entropy loss is defined as:

$$L_{\text{cross-entropy}} = -\sum_i y_i \log(\hat{y}_i)$$

   For the correct class (where $y_i = 1$),to minimize cross-entropy loss, $\hat{y}_i$ must approach 1. The term $y_i \log(\hat{y}_i)$ becomes smaller as $\hat{y}_i$ gets closer to 1, reducing the negative log term. For the incorrect classes (where $y_i = 0$),to minimize the loss, $\hat{y}_i$ must approach 0. In this case, $y_i = 0$, so the corresponding $y_i \log(\hat{y}_i)$ terms are zeroed out regardless of $\hat{y}_i$. However, if $\hat{y}_i$ is far from zero for incorrect classes, it indicates that the model is incorrectly assigning too much probability to the wrong classes, increasing the loss.

   For a single example, the squared error loss is defined as:

$$L_{\text{squared error}} = \frac{1}{2} \sum_i (\hat{y}_i - y_i)^2$$

To minimize the squared error loss, $\hat{y}_i$ must approach y_i for each class. For the correct class (where $y_i = 1$), $\hat{y}_i$ should move closer to 1. For the incorrect classes (where $y_i = 0$), $\hat{y}_i$ should move closer to 0. In squared error loss, the difference $(\hat{y}_i - y_i)^2$ is minimized when $\hat{y}_i$ matches y_i, regardless of the exact form of the output (binary or continuous).

9. How are these functions better suited to classification or regression tasks?
   Answer: Cross-Entropy Loss is more suitable for classification tasks because it is specifically designed to work with probabilities (i.e., SoftMax outputs) and one-hot encoded targets. It is more sensitive when the network is confident but wrong, leading to larger gradients and better training dynamics in classification tasks.
   Squared Error Loss is more general and can be used for regression or classification but doesn't handle probabilities as naturally as cross-entropy. It treats the difference between predicted and actual values symmetrically, which can be less effective when dealing with probability distributions, particularly for classification tasks.

2.4 Optimization algorithm

10. What seem to be the advantages and disadvantages of the various variants of gradient descent between the classic, mini-batch stochastic and online stochastic versions? Which one seems the most reasonable to use in the general case?
    Answer: **Gradient Descent:**
    Advantages:

    (a) Gradient descent uses the average gradient of the entire dataset, resulting in **more accurate** updates to the model's parameters.

    (b) Unlike SGD, gradient descent has a more **predictable convergence behaviour**, making it easier to fine-tune the learning rate and other hyperparameters.

    (c) Because it uses the average gradient, gradient descent is **less sensitive to noisy data** than SGD.

    Disadvantages:

    (a) Gradient descent requires calculating the gradient for the entire dataset in each iteration, making it much **slower** than SGD, especially for large datasets.

    (b) Due to the need to store the entire dataset for each iteration, gradient descent requires **more memory** than SGD.

    Mini-batch Stochastic Gradient:
    Advantages:

    (a) The model update frequency is higher than batch gradient descent which allows for a more robust convergence, avoiding local minima.

    (b) The batched updates provide a computationally more efficient process than stochastic gradient descent.

    (c) The batching allows both the efficiency of not having all training data in memory and algorithm implementations.

    Disadvantages:

    (a) Mini-batch requires the configuration of an additional "mini-batch size" hyperparameter for the learning algorithm.

(b) Error information must be accumulated across mini-batches of training examples like batch gradient descent.

Online Stochastic Gradient Descent:
Advantages:

(a) It updates the model parameters after each example, making it very efficient for large-scale datasets.

(b) It quickly learns from data since parameter updates are immediate, which can be useful in real-time or streaming applications.

Disadvantages:

(a) The gradient calculated on a single example is highly noisy, leading to unstable updates and fluctuating convergence. It may take many iterations to find the optimal solution.

(b) Due to the noisy updates, online SGD can overshoot the minimum of the loss function, especially with a poorly chosen learning rate.

(c) Since the parameters are updated for each individual example, it might become inefficient for smaller datasets where full-batch or mini-batch methods would converge faster.

In most cases, mini-batch stochastic gradient descent strikes the best balance between the three methods. It combines the computational efficiency of stochastic methods with the stability of full-batch methods. It provides faster convergence than full-batch gradient descent, especially for large datasets. It also is more stable than online SGD, with smaller fluctuations in the gradient, leading to smoother optimization.

11. What is the influence of the learning rate η on learning?
Answer: The learning rate η controls the step size when updating the model's parameters in the direction of the gradient. A large learning rate causes larger steps in the direction of the gradient, leading to faster initial learning in some cases and skipping the global or local minimum, thus failing to converge. In contrast to this, a small learning rate causes very small steps in the direction of the gradient. While this may ensure that the optimization process moves towards the minimum more cautiously, it also results in increasing risks of getting stuck in a local minima or a very slow convergence, which in turn increases the computational costs a lot. A moderate or adaptive learning rate will allow the model to make steady progress towards the minimum of the loss function without oscillating or diverging. It will balance speed and accuracy in optimization.

12. Compare the complexity (depending on the number of layers in the network) of calculating the gradients of the loss with respect to the parameters, using the naive approach and the *backprop* algorithm.
Answer:
Naive Approach: Here, the gradient of the loss function is calculated independently for each layer, without reusing computations from previous layers. This involves calculating the derivative of the loss with respect to each parameter in the network by applying the chain rule repeatedly at each layer. For each layer, the forward and backward passes requires $O(n_i \times n_{i-1})$ operations, where n_i is the number of neurons in the $i-th$ layer. Since each layer's gradient computation doesn't reuse results from the previous layer, the total complexity is multiplied by L, the number of layers in the network , resulting in a total complexity of $O(L^2)$ for computing the gradient over all layers.
Backpropagation algorithm approach: Here, first, the forward pass is performed

through the network, computing the activations and storing them for each layer. For each layer, this still requires $O(n_i \times n_{i-1})$ operations, as in the naive approach. But during the backward pass, the gradient with respect to each parameter in layer i can be calculated by reusing the previously computed gradient from layer $i+1$. This avoids repeating the full gradient calculation for every parameter at each layer. The forward pass has a complexity of $O(L)$, as we compute activations for each layer sequentially, but the backward pass also has a complexity of $O(L)$, since the gradient at each layer reuses the gradient from the previous layer and propagates it backward. Thus, the total complexity of backpropagation is $O(L)$, which grows linearly with the number of layers.
This quadratic growth in complexity makes the naive approach inefficient as the number of layers increases.

13. What criteria must the network architecture meet to allow such an optimization procedure ?
Answer:

    (a) The activation and loss functions used in the network must be differentiable since the backpropagation relies on calculating gradients of the loss function with respect to the model's parameters using the chain rule.

    (b) Functions must be continuous to avoid sudden changes in gradients.

    (c) Proper initialization of weights is essential for avoiding problems with gradient flow and 'dead' neurons.

    (d) The network must have parameters (weights and biases) that can be updated during training.

    (e) Inclusion of an objective function model that can be optimized. This objective function can have arbitrary architectures and activation functions, allowing for flexibility in the optimization process.

14. The function SoftMax and the loss of cross-entropy are often used together and their gradient is very simple. Show that the loss can be simplified by: $l = -\sum_i y_i \tilde{y_i} + \log\left(\sum_i e^{\tilde{y_i}}\right)$
Answer: We know that for a given vector $\tilde{y}$, the SoftMax function for the $i-th$ class is defined as:

$$\hat{y}_i = \frac{e^{\tilde{y}_i}}{\sum_j e^{\tilde{y}_j}} \leftarrow (1)$$

and the cross-entropy loss between the true label vector y and the predicted probability vector $\hat{y}$ is defined as:

$$l = -\sum_i y_i \log(\hat{y}_i)$$

Now, substituting the value of $\hat{y}_i$ from Eq. 1:

$$l = -\sum_i y_i \log\left(\frac{e^{\tilde{y}_i}}{\sum_j e^{\tilde{y}_j}}\right)$$

$$=> l = -\sum_i y_i \left(\log\left(e^{\tilde{y}_i}\right) - \log\left(\sum_j e^{\tilde{y}_j}\right)\right)$$

$$=> l = -\sum_i y_i \left[\tilde{y}_i - \log\left(\sum_j e^{\tilde{y}_j}\right)\right]$$

$$=> l = -\sum_i y_i \tilde{y}_i + \sum_i y_i \log\left(\sum_j e^{\tilde{y}_j}\right)$$

Now, since $\sum_i y_i = 1$ (because y is a one-hot encoded vector), the above expressions becomes:

$$l = -\sum_i y_i \tilde{y}_i + \log\left(\sum_j e^{\tilde{y}_j}\right)$$

Hence, proved.

15. Write the gradient of the loss (cross-entropy) relative to the intermediate output $\tilde{y}$.
Answer: In the previous answer we saw that loss can be written as:

$$l = -\sum_i y_i \tilde{y}_i + \log\left(\sum_j e^{\tilde{y}_j}\right)$$

Now, the gradient of the first term with respect to $\tilde{y}_i$ is:

$$\frac{\partial}{\partial \tilde{y}_i}\left(-\sum_j y_j \tilde{y}_j\right) = -y_i$$

The gradient of the second term is calculated using the chain rule as follows:

$$\frac{\partial}{\partial \tilde{y}_i} \log\left(\sum_j e^{\tilde{y}_j}\right) = \frac{1}{\sum_j e^{\tilde{y}_j}} \cdot \frac{\partial}{\partial \tilde{y}_i}\left(\sum_j e^{\tilde{y}_j}\right)$$

$$= \frac{e^{\tilde{y}_i}}{\sum_j e^{\tilde{y}_j}} = \hat{y}_i,$$

where $\hat{y}_i$ is the SoftMax output for class i (the predicted probability for class i).

The total gradient of the loss with respect to $\tilde{y}_i$ is the sum of the gradients of the two terms:

$$\frac{\partial l}{\partial \tilde{y}_i} = \hat{y}_i - y_i$$

We can now express the full gradient with respect to the logits $\tilde{y}$ in vector form as:

$$\nabla_{\tilde{y}} l = \begin{bmatrix} \frac{\partial l}{\partial \tilde{y}_1} \\ \frac{\partial l}{\partial \tilde{y}_2} \\ \vdots \\ \frac{\partial l}{\partial \tilde{y}_{n_y}} \end{bmatrix} = \begin{bmatrix} \hat{y}_1 - y_1 \\ \hat{y}_2 - y_2 \\ \vdots \\ \hat{y}_{n_y} - y_{n_y} \end{bmatrix}$$

16. Using the *backpropagation*, write the gradient of the loss with respect to the weights of the output layer $\nabla_{W_y} l$. Note that writing this gradient uses $\nabla_{\tilde{y}} l$. Do the same for $\nabla_{b_y} l$.
Answer: $\nabla_{W_y,ij} l = \frac{\partial l}{\partial W_y} = \Sigma_{k=1}^{n_y} \frac{\partial l}{\partial \tilde{y}_k} \cdot \frac{\partial \tilde{y}_k}{\partial W_{y,ij}}$
Now, we already know that $\nabla_{\tilde{y}} l = \hat{y}_i - y_i$ from the last question.

So, now we evaluate the differentiation of the 2nd term of the product.

$\frac{\partial \tilde{y}_k}{\partial W_y,ij} = \frac{\partial [W_y h + b_y]_k}{\partial W_y,ij} = \frac{\partial [W_y h]_k}{\partial W_y,ij}$

$\frac{\partial (\Sigma_{p=1}^{n_h} W_y,iph_p)}{\partial W_y,ij} = h_j$, if $k = i; 0$, if $k \neq i$.

Thus, $\frac{\partial l}{\partial W_y} = (\hat{y}_i - y_i).h_j$

$\Rightarrow \nabla_{W_y} l = (\hat{y}_i - y_i).h^T$

$$\Rightarrow \nabla_{W_y} l = \begin{pmatrix} (\hat{y}_1 - y_1)h_1 & (\hat{y}_1 - y_1)h_2 & \dots & (\hat{y}_1 - y_1)h_{n_h} \\ (\hat{y}_2 - y_2)h_1 & (\hat{y}_2 - y_2)h_2 & \dots & (\hat{y}_2 - y_2)h_{n_h} \\ \vdots & \vdots & \ddots & \vdots \\ (\hat{y}_{n_y} - y_{n_y})h_1 & (\hat{y}_{n_y} - y_{n_y})h_2 & \dots & (\hat{y}_{n_y} - y_{n_y})h_{n_h} \end{pmatrix}$$

Now, computing the gradient of the loss with respect to the bias of the output layer,

$\nabla_{b_y} l = \Sigma_{k=1}^{n_y} \frac{\partial l}{\partial \tilde{y}_k} . \frac{\partial \tilde{y}_k}{\partial b_y}$

Determining the second part of the above product

$\frac{\partial \tilde{y}_k}{\partial b_y,i} = \frac{\partial [W_y h + b_y]_k}{\partial b_y,i} = \frac{\partial [b_y]_k}{\partial b_y,i} = 1$, if $k = i; 0$, if $k \neq i$.

Therefore, $\nabla_{b_y} l = (\hat{y}_i - y_i).1 = (\hat{y}_i - y_i)$

$$\Rightarrow \nabla_{b_y} l = \begin{pmatrix} \hat{y}_1 - y_1 \\ \hat{y}_2 - y_2 \\ \vdots \\ \hat{y}_{n_y} - y_{n_y} \end{pmatrix}$$

17. Compute other gradients : $\nabla_{\tilde{h}} l$, $\nabla_{W_h} l$, $\nabla_{b_h} l$.
Answer: $\nabla_{\tilde{h}} l$**:**

$\nabla_{\tilde{h}} l = \frac{\partial l}{\partial \tilde{h}} = \Sigma_{k=1}^{n_y} \frac{\partial l}{\partial \tilde{y}_k} . \frac{\partial \tilde{y}_k}{\partial \tilde{h}}$

Now, we already know that $\nabla_{\tilde{y}} l = \hat{y}_i - y_i$ from the previous solutions.

So, now we evaluate the differentiation of the 2nd term of the product.

$\frac{\partial \tilde{y}_k}{\partial \tilde{h}} = \frac{\partial [W_y h + b_y]_k}{\partial \tilde{h}} = \frac{\partial [W_y h]_k}{\partial \tilde{h}}$

$\frac{\partial (\Sigma_{p=1}^{n_h} W_y,iph_p)}{\partial W_y,ij} = W_y, j$, if $k = i; 0$, if $k \neq i$.

Thus, $\frac{\partial l}{\partial \tilde{h}} = W_y, j.(\hat{y}_i - y_i)$

$\Rightarrow \nabla_{\tilde{h}} l = W^T.(\hat{y}_i - y_i)$

$$\Rightarrow \nabla_{\tilde{h}} l = \begin{pmatrix} w_{11} & w_{12} & \dots & w_{1n_h} \\ w_{21} & w_{22} & \dots & w_{2n_h} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n_y 1} & w_{n_y 2} & \dots & w_{n_y n_h} \end{pmatrix}^\top \begin{pmatrix} \hat{y}1 - y_1 \\ \hat{y}2 - y_2 \\ \vdots \\ \hat{y}n_y - yn_y \end{pmatrix}$$

$\nabla_{W_h} l$:

$\nabla_{W_h,ij} l = \Sigma_{k=1}^{n_h} \frac{\partial l}{\partial \tilde{h}_k} \cdot \frac{\partial \tilde{h}_k}{\partial W_h,ij}$

Since $\tilde{h} = W_h x + b_h$,

$\frac{\partial \tilde{h}}{\partial W_h,ij} = \frac{\partial [W_h x + b_h]_k}{\partial W_h,ij} = \frac{\partial [W_h x]_k}{\partial W_h,ij}$

Now, $\frac{\partial (\Sigma_{p=1}^{n_h} W_h,ipx_p)}{\partial W_h,ij} = x_j$, if $k = i; 0$, if $k \neq i$.

Thus, $\frac{\partial l}{\partial W_h} = W^T.(\hat{y}_i - y_i).x^j$

$\Rightarrow \nabla_{W_y} l = W^T.(\hat{y}_i - y_i).x^T$

$$\Rightarrow \nabla_{W_y} l = \begin{pmatrix} (\nabla_{\tilde{h}} l)_1 x_1 & (\nabla_{\tilde{h}} l)_1 x_2 & \dots & (\nabla_{\tilde{h}} l)_1 x_{n_x} \\ (\nabla_{\tilde{h}} l)_2 x_1 & (\nabla_{\tilde{h}} l)_2 x_2 & \dots & (\nabla_{\tilde{h}} l)_2 x_{n_x} \\ \vdots & \vdots & \ddots & \vdots \\ (\nabla_{\tilde{h}} l) n_h x_1 & (\nabla_{\tilde{h}} l)_{n_h} x_2 & \dots & (\nabla_{\tilde{h}} l) n_h x_{n_x} \end{pmatrix}$$

$\nabla_{b_h} l$:

The gradient of the loss with respect to b_h is:

$\nabla_{b_h} l = \nabla_{\tilde{h}} l$

This is simply the gradient with respect to the affine transformation before applying the activation, and it is a vector of size n_h .

In vector form:

$$\nabla_{b_h} l = \begin{pmatrix} (\nabla_{\tilde{h}} l)_1 \\ (\nabla \tilde{h} l)_2 \\ \vdots \\ (\nabla \tilde{h} l)_{n_h} \end{pmatrix}$$

3 Implementation

Done in the python notebook.

4 Conclusion

Through this lab, we achieved a working model that accurately performs forward and backward passes, calculates loss, and updates parameters. These results reinforce our understanding of each step involved in neural network training. The practical experience with parameter initialization, gradient computation, and stochastic gradient descent provides insight into how neural networks learn from data, laying the foundation for building and training more complex models in future studies.