Neural Networks

Sankalp Vats

May 2025

1 Introduction

2 Neural Network from Scratch

2.1 Introduction

2.2 Dataset Description

2.3 Neural Network Architecture

2.4 Forward Propagation

2.5 Loss Function

2.6 Backpropagation Algorithm

2.7 Gradient Descent Optimization

2.8 Training and Evaluation

2.9 Challenges and Debugging

2.10 Summary and Insights

3 Neural Network using PyTorch

3.1 Introduction

3.2 Dataset Loading and Preprocessing

3.3 Model Architecture in PyTorch

3.4 Loss Function and Optimizer

3.5 Training Procedure

3.6 Evaluation and Metrics

3.7 Comparison with Scratch Implementation

3.8 Challenges Faced

3.9 Summary and Learnings

1 Introduction

Neural networks are machine learning models that mimic the complex functions of the human brain. These models consist of interconnected nodes or neurons that process data, learn patterns, and enable tasks such as pattern recognition and decision-making.

In this article, we will explore the fundamentals of neural networks, their architecture, how they work, and their applications in various fields. Understanding neural networks is essential for anyone interested in the advancements of artificial intelligence.

Understanding Neural Networks in Deep Learning Neural networks are capable of learning and identifying patterns directly from data without pre-defined rules. These networks are built from several key components:

Neurons: The basic units that receive inputs, each neuron is governed by a threshold and an activation function. Connections: Links between neurons that carry information, regulated by weights and biases. Weights and Biases: These parameters determine the strength and influence of connections. Propagation Functions: Mechanisms that help process and transfer data across layers of neurons. Learning Rule: The method that adjusts weights and biases over time to improve accuracy. Learning in neural networks follows a structured, three-stage process:

Input Computation: Data is fed into the network. Output Generation: Based on the current parameters, the network generates an output. Iterative Refinement: The network refines its output by adjusting weights and biases, gradually improving its performance on diverse tasks. In an adaptive learning environment:

The neural network is exposed to a simulated scenario or dataset. Parameters such as weights and biases are updated in response to new data or conditions. With each adjustment, the network's response evolves, allowing it to adapt effectively to different tasks or environments. Types of Neural Networks There are seven types of neural networks that can be used.

Feedforward Networks: A feedforward neural network is a simple artificial neural network architecture in which data moves from input to output in a single direction. Singlelayer Perceptron: A single-layer perceptron consists of only one layer of neurons . It takes inputs, applies weights, sums them up, and uses an activation function to produce an output. Multilayer Perceptron (MLP): MLP is a type of feedforward neural network with three or more layers, including an input layer, one or more hidden layers, and an output layer. It uses nonlinear activation functions. Convolutional Neural Network (CNN): A Convolutional Neural Network (CNN) is a specialized artificial neural network designed for image processing. It employs convolutional layers to automatically learn hierarchical features from input images, enabling effective image recognition and classification. Recurrent Neural Network (RNN): An artificial neural network type intended for sequential data processing is called a Recurrent Neural Network (RNN). It is appropriate for applications where contextual dependencies are critical, such as time series prediction and natural language processing, since

it makes use of feedback loops, which enable information to survive within the network. Long Short-Term Memory (LSTM): LSTM is a type of RNN that is designed to overcome the vanishing gradient problem in training RNNs. It uses memory cells and gates to selectively read, write, and erase information.

2 Neural Network From Scratch

2.1 Introduction

Neural networks are a core component of deep learning models, and implementing them from scratch is a great way to understand their inner workings. we will demonstrate how to implement a basic Neural networks algorithm from scratch using the NumPy library in Python.However,we will be using some libraries like scikit-learn,pandas also for efficient data cleaning so that it fits well to our neural network.

2.2 Dataset Description

The dataset used in this project is the **Medical Appointment No-Show Dataset**, obtained from Kaggle. It contains information on over 110,000 medical appointments in Brazil, with a primary focus on whether or not a patient showed up for their scheduled appointment. The dataset is used for binary classification: predicting whether a patient will attend the appointment (`No-show = No`) or miss it (`No-show = Yes`).

Key features of the dataset include:

- **PatientId, AppointmentID**: Unique identifiers for each patient and appointment.
- **Gender**: Indicates the gender of the patient.
- **ScheduledDay**: The date and time the appointment was scheduled.
- **AppointmentDay**: The actual date of the appointment.
- **Age**: Age of the patient.
- **Neighbourhood**: The location of the hospital.
- **Scholarship**: Whether the patient is enrolled in the Brazilian welfare program Bolsa Família.
- **Hypertension, Diabetes, Alcoholism, Handcap**: Indicate the presence of various health conditions.
- **SMS_received**: Indicates whether an appointment reminder SMS was sent to the patient.
- **No-show**: Target variable indicating whether the patient missed the appointment.

Preprocessing Steps: Before training the model, several preprocessing steps were applied:

- Removal of irrelevant columns such as `PatientId` and `AppointmentID`.

- Conversion of categorical features (e.g., `Gender`, `Neighbourhood`) into numerical representations using one-hot encoding or label encoding.
- Parsing date features to extract useful information such as the waiting time between scheduling and the appointment date.
- Normalization of continuous variables (e.g., age) for improved training convergence.
- Handling class imbalance if needed, since the dataset has more "show" records than "no-show".

This dataset is well-suited for binary classification problems and provides a real-world scenario to test both manually implemented and framework-based neural network models.

2.3 Neural Network Architecture

The neural network was built completely from scratch using only NumPy, without relying on any machine learning or deep learning frameworks. The architecture is a simple yet powerful feedforward neural network designed for binary classification on the Medical Appointment No-Show Dataset.

The network consists of the following components:

- **Input Layer:** The number of input features is determined after preprocessing the dataset. Features like gender, age, scholarship status, and encoded neighbourhood information are used. These inputs are standardized using `StandardScaler` to ensure stable training.
- **First Dense Layer (Hidden Layer):** This is a fully connected layer implemented as the `Layer_Dense` class. It takes the input vector and connects it to 64 neurons. The weights are initialized with small random values (scaled by 0.01) and biases are initialized to zero. Mathematically, this layer computes:
$$z = X \cdot W + b$$
- **ReLU Activation Layer:** A ReLU (Rectified Linear Unit) activation function follows the first dense layer. ReLU is applied element-wise as:
$$ReLU(x) = \max(0, x)$$
This introduces non-linearity and allows the network to learn complex patterns in the data.
- **Second Dense Layer (Output Layer):** This layer connects the 64-dimensional hidden representation to 2 output neurons, which represent the two target classes: 'No-show' or 'Show'. The weights and biases are again initialized randomly and set to zero, respectively.

- **Softmax Activation and Loss Combination:** The final layer output is passed through a Softmax activation function, converting the raw scores into probabilities for each class. A combined class called `Activation_Softmax_Loss` performs both the Softmax transformation and the categorical cross-entropy loss calculation in one step for numerical stability.

- **Loss Function:** Categorical cross-entropy is used to measure how well the predicted probabilities match the actual labels. It is defined as:

$$L = -\log(p_y)$$

  where p_y is the predicted probability for the correct class.

- **Backward Pass (Gradient Calculation):** The network supports backpropagation by implementing custom `backward()` methods in each layer. Gradients are calculated layer by layer in reverse order, starting from the loss, and propagated backward through the dense and activation layers.

- **Weight Updates (Gradient Descent):** Weights and biases are updated using basic gradient descent. The learning rate is set to 0.01, and updates follow the rule:

$$W := W - \eta \cdot \frac{\partial L}{\partial W} \quad and \quad b := b - \eta \cdot \frac{\partial L}{\partial b}$$

  where η is the learning rate.

The model is trained for 1000 epochs using a manual training loop. After each forward and backward pass, the network parameters are updated. Every 100 epochs, the loss and accuracy on the training data are printed to monitor learning progress. The overall design demonstrates a foundational understanding of deep learning principles such as activation functions, loss calculation, gradient propagation, and optimization.

2.4 Forward Propagation

Forward propagation is the process through which input data is passed through the neural network to generate predictions. It involves computing the output of each layer sequentially from the input layer to the output layer. In our implementation, forward propagation is structured into clearly defined stages:

1. **Input Layer to First Dense Layer:** The input feature matrix X is multiplied with the weight matrix $W^{(1)}$ of the first dense layer and added to its bias vector $b^{(1)}$. This operation produces the pre-activation output:

$$Z^{(1)} = X \cdot W^{(1)} + b^{(1)}$$

   This output is passed to the next layer.

2. **ReLU Activation:** The result from the first dense layer is then passed through a ReLU (Rectified Linear Unit) activation function, which applies a non-linear transformation. ReLU sets all negative values to zero, maintaining positive values as they are:

$$A^{(1)} = ReLU(Z^{(1)}) = \max(0, Z^{(1)})$$

3. **Second Dense Layer:** The ReLU-activated output is fed into the second fully connected layer. A similar linear transformation is applied using the second layer's weights $W^{(2)}$ and biases $b^{(2)}$:

$$Z^{(2)} = A^{(1)} \cdot W^{(2)} + b^{(2)}$$

4. **Softmax Activation (Output Layer):** Finally, the network uses the Softmax function to transform the raw output scores $Z^{(2)}$ into probabilities. Softmax ensures that the outputs are positive and sum to one, making them interpretable as class probabilities:

$$\hat{y}_i = \frac{e^{Z_i^{(2)}}}{\sum_j e^{Z_j^{(2)}}}$$

   where $\hat{y}_i$ represents the predicted probability of class i for a given input sample.

This sequence completes the forward pass and results in a probability distribution over the target classes for each input sample. These predicted probabilities are then used to compute the loss and guide the network's learning during backpropagation.

2.5 Loss Function

To quantify how well the neural network's predictions align with the actual target labels, the **categorical cross-entropy loss function** was employed. This loss function is particularly appropriate for multi-class classification tasks and works synergistically with the Softmax activation used in the output layer.

The categorical cross-entropy loss is defined as:

$$\mathcal{L} = -\sum_{i=1}^{N} \log(\hat{y}_i)$$

where $\hat{y}_i$ is the predicted probability of the true class label for the i-th sample, and N is the number of samples. For numerical stability, predictions are clipped to avoid taking the logarithm of zero. The loss reflects the negative log-likelihood of the correct class, penalizing confident but incorrect predictions more heavily.

2.6 Backpropagation Algorithm

The **backpropagation algorithm** was implemented to compute gradients of the loss function with respect to all learnable parameters — specifically, the weights and biases of the dense layers. These gradients are essential for updating the model in a direction that reduces prediction error.

The process is based on the *chain rule of calculus* and flows in reverse order through the network:

1. The combined Softmax and categorical cross-entropy layer computes the initial gradient of the loss with respect to the output logits.
2. This gradient is propagated backward through the second dense layer, calculating gradients for its weights, biases, and inputs.
3. The gradient then flows through the ReLU activation, where it is element-wise masked to zero for inputs that were non-positive during the forward pass.
4. Finally, the gradient continues to the first dense layer, where gradients for its weights and biases are computed similarly.

Each layer has its own `backward()` method which encapsulates the logic needed to compute partial derivatives with respect to its inputs and parameters.

2.7 Gradient Descent Optimization

To update the model parameters, the **gradient descent optimization** technique was employed. This method adjusts the weights and biases in the opposite direction of the gradient of the loss function with respect to those parameters:

$$W := W - \eta \cdot \frac{\partial \mathcal{L}}{\partial W}, \quad b := b - \eta \cdot \frac{\partial \mathcal{L}}{\partial b}$$

where η is the learning rate, which was set to a fixed value of 0.01 in this project. This learning rate controls how much the parameters are updated during each iteration. Over the course of training, repeated updates gradually reduce the loss, enabling the network to improve its classification performance on unseen data.

2.8 Training and Evaluation

The neural network was trained over **1000 epochs**. Each epoch involved a complete forward pass through the network, calculation of loss and accuracy, followed by a backward pass to compute gradients, and finally, an update of weights and biases using gradient descent.

During the forward pass:

- Inputs were passed through two dense (fully connected) layers.

- The ReLU activation was applied after the first layer.
- The output of the final layer was passed through a Softmax function to obtain class probabilities.
- The combined Softmax and categorical cross-entropy layer computed the loss and initiated backpropagation.

After each epoch, accuracy was computed by comparing the predicted class (the index of the highest Softmax probability) to the true labels. The loss and accuracy were printed every 100 epochs to monitor convergence and training progress.

2.9 Challenges and Debugging

Several technical challenges were encountered during development:

- **Backpropagation correctness:** Ensuring that gradients were propagated accurately through both activation and dense layers was critical. Errors in gradient computation often led to exploding or vanishing loss values.
- **Shape mismatches:** Care was taken to maintain dimensional consistency across matrix operations. Debugging involved printing shapes of inputs, weights, outputs, and gradients at each layer.
- **Numerical stability:** During Softmax computation, large exponentials could lead to instability. This was addressed by shifting logits before applying the exponential function, and clipping predictions in the loss function to avoid $\log(0)$ errors.
- **Learning rate tuning:** A fixed learning rate of 0.01 was selected after empirical observation. Larger values caused divergence, while smaller ones resulted in slow convergence.

2.10 Summary and Insights

Implementing a neural network entirely from scratch—without relying on deep learning frameworks—provided a comprehensive understanding of the internal mechanisms of deep learning. Key takeaways include:

- The role of matrix multiplication in propagating data through layers.
- How activation functions like ReLU affect gradient flow.
- The sensitivity of training to hyperparameters and numerical precision.
- The interplay between loss computation and output layer design.

This project reinforced foundational concepts such as forward propagation, backpropagation, gradient descent, and classification loss functions. It also highlighted the importance of modular, well-tested components when building machine learning systems. Despite being a relatively simple architecture, this exercise deepened conceptual clarity and provided practical experience with core algorithms underpinning modern neural networks.

3 Neural Network Using Pytorch

3.1 Introduction

PyTorch is a popular open-source framework used for building and training neural networks. It was developed by Facebook's AI Research lab and has gained wide acceptance due to its flexibility and ease of use. PyTorch allows programmers to write neural network models in Python, making it accessible even for beginners while still powerful for experts.

One of the key advantages of PyTorch is its dynamic computation graph, which means the network architecture can be changed on the fly during training. This is different from other frameworks that use static graphs and makes PyTorch easier to debug and modify. Additionally, PyTorch supports automatic differentiation, so it automatically computes gradients needed for optimizing the network parameters through backpropagation.

Using PyTorch simplifies many complex tasks such as handling matrix operations, gradient calculations, and running code on GPUs for faster training. This lets developers focus more on designing and improving their models rather than on low-level implementation details. In this section, we will discuss the basics of PyTorch and how it helps in efficiently creating, training, and evaluating neural networks.

3.2 Dataset Loading and Preprocessing

Before training a neural network, it is essential to prepare the dataset properly. In PyTorch, datasets can be loaded in various formats, including CSV files, images, or built-in datasets. For this project, the medical appointment no-show dataset was used, which contains information about patient appointments and whether they showed up or not. The first step is to load the dataset using libraries like Pandas, which allows easy reading and manipulation of tabular data.

After loading the data, preprocessing is necessary to convert the raw data into a form suitable for the neural network. This includes handling missing values by either removing or filling them, converting categorical variables such as gender into numerical values using techniques like label encoding, and normalizing continuous features to ensure all values fall within a similar scale. Normalization helps the model learn more efficiently by preventing features with large values from dominating the training process.

In addition, some features may be transformed using one-hot encoding, especially when they represent categories with multiple possible values, such as neighborhood names. This converts each category into a binary vector, allowing the network to treat each category independently.

Finally, the dataset is split into training and test sets. The training set is used to teach the model, while the test set evaluates how well the model generalizes to unseen data. This step is crucial to prevent overfitting, where the model performs well on training data but poorly on new inputs. By carefully

loading and preprocessing the dataset, the neural network receives clean and meaningful input, which improves its learning and prediction abilities.

3.3 Model Architecture in PyTorch

The model architecture defines how the neural network is structured, including the number of layers, neurons in each layer, and the activation functions used. In PyTorch, defining a model is straightforward using the `nn.Module` class. For this project, a simple feedforward neural network was built with two dense (fully connected) layers.

The first layer takes the input features and maps them to a hidden layer with 64 neurons. This layer is followed by a ReLU (Rectified Linear Unit) activation function, which introduces non-linearity to the model. Non-linear activation functions allow the network to learn complex patterns in the data rather than just simple linear relationships.

The second dense layer takes the output from the hidden layer and produces predictions for the two classes: whether a patient will show up or not. This output is passed through a Softmax function, which converts raw output scores into probabilities that sum to one, making them interpretable as class likelihoods.

PyTorch makes it easy to define the forward pass through the network, where data flows from the input through each layer and activation function to the output. This modular approach simplifies experimentation with different architectures, such as adding more layers, changing the number of neurons, or using different activation functions. The chosen architecture balances complexity and performance to provide accurate predictions without excessive computation.

3.4 Loss Function and Optimizer

The loss function is a key component in training neural networks, as it quantifies how well the model's predictions match the actual labels. For classification tasks like this one, the categorical cross-entropy loss is commonly used. This loss measures the difference between the predicted probability distribution (from the Softmax output) and the true class labels. A lower loss indicates better performance, guiding the model towards improved accuracy.

In PyTorch, loss functions are provided in the `torch.nn` module, making it easy to incorporate them into the training loop. The categorical cross-entropy loss (also called `CrossEntropyLoss`) automatically combines the Softmax activation and the cross-entropy calculation for efficiency.

The optimizer updates the model parameters (weights and biases) based on the gradients computed from the loss. Gradient descent is the basic optimization method used here, which moves the parameters in the direction that reduces the loss. PyTorch offers many optimization algorithms; in this case, the stochastic gradient descent (SGD) optimizer is used with a fixed learning rate of 0.01. This learning rate controls the step size taken during each update.

By iteratively computing the loss, calculating gradients through backpropagation, and updating parameters using the optimizer, the model gradually

improves its ability to predict the correct class. Choosing the right loss function and optimizer, as well as tuning their parameters, is essential to training an effective neural network.

3.5 Training Procedure

Training the neural network involves repeatedly presenting the data to the model and adjusting its parameters to minimize the loss. In PyTorch, this is done using a loop called the training loop. Each iteration of the loop is called an epoch, and during each epoch, the entire training dataset is passed forward through the model to generate predictions.

First, the input data is fed into the model's forward function, which computes outputs based on the current weights and biases. The predicted outputs are then compared with the true labels using the loss function to calculate how far off the predictions are. Next, PyTorch's automatic differentiation engine computes gradients of the loss with respect to each parameter in the network by backpropagation. These gradients indicate the direction and amount by which the parameters should be adjusted.

After calculating gradients, the optimizer updates the model's parameters, typically by subtracting a small portion of the gradients scaled by the learning rate. This step moves the model closer to the optimal parameters that minimize the loss. The training process repeats this forward and backward pass over many epochs until the model achieves satisfactory accuracy or the loss stabilizes.

To prevent the model from overfitting the training data, techniques such as shuffling the data and using a validation set can be employed. Shuffling helps by changing the order of the data in each epoch, ensuring the model does not learn data order biases. While this basic training procedure was used in this project, more advanced techniques like learning rate scheduling or early stopping can further improve training efficiency.

The training process also involves monitoring the loss and accuracy at regular intervals to observe progress. Printing these metrics every few epochs helps to identify if the model is learning properly or if issues such as vanishing gradients or overfitting are occurring. By carefully following this training procedure, the neural network can learn to make reliable predictions on new, unseen data.

3.6 Evaluation and Metrics

Evaluating the trained neural network is crucial to understand how well it performs on unseen data. The evaluation is usually done using a separate test dataset that the model has never seen during training. This helps measure the model's ability to generalize beyond the training data.

The primary metric used in this classification task is accuracy, which is the ratio of correctly predicted samples to the total number of samples. While accuracy is easy to understand, it may not always fully capture the model's performance, especially when dealing with imbalanced datasets, where one class occurs much more frequently than the other.

In addition to accuracy, other metrics like precision, recall, and F1-score can provide deeper insight. Precision measures how many of the predicted positive cases are actually positive, while recall measures how many actual positive cases were correctly identified by the model. The F1-score is the harmonic mean of precision and recall, providing a balance between the two.

Confusion matrices are also commonly used to visualize model performance. They show the counts of true positive, true negative, false positive, and false negative predictions, helping to identify specific weaknesses such as false alarms or missed cases.

During evaluation, the model's loss on the test set is also computed to check if it is consistent with the loss observed during training. A large gap between training and test loss might indicate overfitting.

Overall, evaluating the model with multiple metrics gives a more comprehensive view of its strengths and limitations, guiding further improvements and tuning to achieve better predictive performance.

3.7 Comparison with Scratch Implementation

Both the scratch-built neural network and the PyTorch implementation demonstrated similar learning trends and final performance, showcasing the effectiveness of both approaches in solving the classification task. Initially, the scratch model started with a loss of approximately 0.693 and an accuracy close to 50.6%, which improved steadily to a loss of around 0.502 and accuracy of 79.78% by the 900th epoch. In contrast, the PyTorch model began with a higher initial loss of 0.792 and a lower accuracy of about 27.46%, but it improved more consistently and slightly outperformed the scratch model by reaching a loss of approximately 0.464 and an accuracy of 80.29% around the 1200th epoch before training was stopped due to no further improvements.

The PyTorch implementation benefited from optimized tensor operations and efficient backpropagation routines, which allowed for smoother convergence and marginally better accuracy. Despite this, the scratch model's performance was impressively close, confirming that the core concepts of forward propagation, loss calculation, backpropagation, and gradient descent were correctly implemented from scratch. This comparison highlights that while frameworks like PyTorch provide advanced tools for building and training neural networks with greater speed and scalability, understanding the underlying mechanics through manual implementation is valuable for grasping the fundamentals of deep learning.

Overall, both models effectively learned to classify the data, with PyTorch having a slight edge in accuracy and training stability. This underscores the importance of leveraging mature libraries for practical deep learning applications, while foundational implementations serve as excellent learning experiences.

3.8 Comparison with Scratch Implementation

Both the scratch-built neural network and the PyTorch implementation demonstrated similar learning trends and final performance, showcasing the effectiveness of both approaches in solving the classification task. Initially, the scratch model started with a loss of approximately 0.693 and an accuracy close to 50.6%, which improved steadily to a loss of around 0.502 and accuracy of 79.78% by the 900th epoch. In contrast, the PyTorch model began with a higher initial loss of 0.792 and a lower accuracy of about 27.46%, but it improved more consistently and slightly outperformed the scratch model by reaching a loss of approximately 0.464 and an accuracy of 80.29% around the 1200th epoch before training was stopped due to no further improvements.

The PyTorch implementation benefited from optimized tensor operations and efficient backpropagation routines, which allowed for smoother convergence and marginally better accuracy. Despite this, the scratch model's performance was impressively close, confirming that the core concepts of forward propagation, loss calculation, backpropagation, and gradient descent were correctly implemented from scratch. This comparison highlights that while frameworks like PyTorch provide advanced tools for building and training neural networks with greater speed and scalability, understanding the underlying mechanics through manual implementation is valuable for grasping the fundamentals of deep learning.

Overall, both models effectively learned to classify the data, with PyTorch having a slight edge in accuracy and training stability. This underscores the importance of leveraging mature libraries for practical deep learning applications, while foundational implementations serve as excellent learning experiences.

3.9 Challenges Faced

During the development of both the scratch neural network and the PyTorch implementation, several challenges were encountered. One major challenge in the scratch implementation was correctly handling the backward pass and ensuring gradients were computed accurately for each layer and activation function. Implementing backpropagation from first principles required careful application of the chain rule, and debugging gradient shapes and values was often time-consuming. Numerical stability also posed problems, especially in the Softmax activation and cross-entropy loss calculations, which had to be addressed through clipping and exponent shifting to prevent overflow or underflow. On the PyTorch side, while many low-level details were abstracted away, configuring the model, optimizer, and loss function correctly to match the scratch network setup required attention. Another difficulty was ensuring the training process was comparable by adjusting hyperparameters like learning rate and epochs. Additionally, preprocessing the dataset consistently for both implementations was critical to obtain reliable and comparable results. Overall, these challenges reinforced the importance of a systematic approach, careful debugging, and understanding of neural network internals.

3.10 Summary and Learnings

This project provided valuable insights into the fundamentals and practical aspects of neural networks by implementing models both from scratch and using PyTorch. Building the network manually helped develop a deep understanding of core concepts such as forward propagation, activation functions, loss calculations, backpropagation, and gradient descent optimization. It also highlighted the complexities involved in ensuring numerical stability and correctness in the training process. On the other hand, the PyTorch implementation showcased the efficiency, scalability, and ease of use provided by modern deep learning frameworks, enabling faster experimentation and better performance with less code. Comparing both models demonstrated that while frameworks simplify development, foundational knowledge remains essential for troubleshooting and customizing models. Overall, this experience strengthened foundational skills in neural networks, enhanced programming proficiency, and emphasized the balance between