Education Reform and Development, 2026, Volume 8, Issue 8
https://ojs.bbwpublisher.com/index.php/ERD
Online ISSN: 2652-5372
Print ISSN: 2652-5364

Jupyter-Based Course Resource Reconstruction and Interactive Teaching Design for Deep Learning

Le Yang, Lin Liu*

Yunnan Normal University, Kunming 650500, China

***Corresponding author:** Lin Liu, liulinrachel@163.com

Abstract: The Deep Learning course combines substantial theoretical content with intensive practical work. In conventional instruction, theoretical explanations, program code, and execution results are often presented separately, making it difficult for students to establish connections among model principles, code implementation, and application tasks. Using Jupyter Notebook as the instructional environment, this study reconstructs existing lecture slides, programming examples, and laboratory materials into coherent, executable teaching units that integrate theoretical explanations, PyTorch code, execution feedback, and learning tasks. Project-based learning, self-directed inquiry, collaborative learning, tiered tasks, and multidimensional assessment are embedded in the design. A convolutional neural network teaching unit is used to illustrate the organization of convolution principles, code-based verification, network construction, image classification, and parameter exploration. The paper further proposes an assessment framework encompassing knowledge mastery, practical competence, learning processes, collaborative performance, and learning experience, together with a mechanism for continuously refining resources in response to teacher and student feedback. The proposed design may inform the integration of theoretical instruction and programming practice in deep learning and related artificial intelligence courses.

Keywords: Jupyter Notebook; Deep learning course; Course resource reconstruction; Interactive teaching; Project-based learning

Online publication: September 8, 2026

1. Introduction

Deep Learning is a core course in artificial intelligence programs. Its content encompasses tensor operations, loss functions, backpropagation, convolutional neural networks, recurrent neural networks, and pretrained models. The course therefore requires students not only to understand mathematical principles and algorithmic mechanisms, but also to perform data processing, model construction, network training, parameter adjustment, and result interpretation. These features make the course theoretically abstract, conceptually interconnected,

and highly demanding in terms of programming practice.

Conventional classes generally use textbooks, board-based explanations, and slide presentations to introduce concepts, formulas, and network architectures, followed by demonstrations of model implementation in separate program files. This arrangement can separate mathematical formulations, source code, and execution results. Students must switch among different resources and software environments, which makes it difficult to establish timely correspondences among theoretical representations, code implementation, and model behavior. Many deep learning concepts also require hands-on execution and comparative analysis to be fully understood. When classroom instruction relies primarily on complete, prewritten programs and fixed outputs, students may remain at the level of code reproduction, with limited opportunities to modify experimental conditions, analyze differences, and solve problems independently.

Jupyter Notebook enables executable code, narrative text, mathematical expressions, visualizations, and computational outputs to be integrated within a single document [1], providing a technical foundation for the continuous organization of theoretical explanation, programming operations, and result feedback. Notebook files can also be saved, modified, and shared, making them suitable for classroom demonstrations, individual practice, project tasks, and records of the learning process. However, simply transferring lecture slides and complete programs into a Notebook does not constitute course resource reconstruction. The resources must be reorganized around knowledge relationships and model execution processes, with project tasks, self-directed inquiry, collaborative interaction, and assessment feedback incorporated into the design.

Accordingly, this paper reconstructs the existing resources of the Deep Learning course in Jupyter Notebook and designs interactive teaching activities on the basis of the reconstructed resources. It focuses on the organization of course content, executable resources, and teaching activities; illustrates the design through a convolutional neural network unit; and proposes a subsequent evaluation and improvement plan. Because full empirical classroom evaluation has not yet been completed, the paper concentrates on course resource development and teaching design and does not make conclusive claims regarding instructional effectiveness.

2. Research background

2.1. Applications of Jupyter Notebook in computational courses

International studies have applied Jupyter Notebook in computer science, data analytics, and engineering courses. Amoudi and Tbaishat organized learning objectives, conceptual explanations, coding tasks, and reflective activities within interactive Notebooks [2]. Bascuñana et al. integrated disciplinary knowledge, numerical computation, and problem-solving activities into chemical engineering teaching modules [3]. Al-Gahmi et al. noted that Notebooks can support not only code demonstrations but also the organization of course content and practical activities [4]. Studies in China have likewise explored the use of Jupyter to integrate theoretical instruction, programming exercises, and project tasks in artificial intelligence courses [5]. These studies indicate that Jupyter can technically support the integrated presentation of theoretical content, programming operations, and computational results. Nevertheless, the division of knowledge units, sequencing of code, task design, and feedback mechanisms still need to be tailored to the characteristics of individual courses.

2.2. Current state of teaching reform in deep learning courses

Existing reforms in Deep Learning courses have primarily focused on integrating theory and practice, revising

course content, developing teaching cases, and strengthening practical competence. Wei and Xiao proposed an integrated theory-practice approach to curriculum and textbook development [6], while Yang et al. strengthened the connection between course knowledge and authentic tasks through representative industry cases [7]. These studies suggest that course reform should go beyond increasing laboratory hours or adding programming examples. It should establish a continuous relationship among model principles, code implementation, and problem solving, while providing students with opportunities for active operation, result comparison, analytical reasoning, and reflection.

Overall, research on Jupyter-based teaching has often examined the use of the platform in individual courses or specific modules, whereas reform studies on Deep Learning courses have mainly addressed curriculum systems, application cases, and practical instruction. Further investigation is therefore needed into how existing lecture slides, programming examples, experimental tasks, and execution results can be reconstructed as coherent, executable Notebook resources, and how project-based inquiry, collaborative learning, and multidimensional assessment can be organized on this basis.

3. Jupyter-based course resource reconstruction

Course resource reconstruction begins with existing lecture slides, programming examples, and teaching content, and reorganizes dispersed theoretical explanations, code, experimental tasks, and execution results into coherent Notebook-based teaching resources. The reconstruction centers on the integrated organization of teaching content and the interactive use of executable resources. Tiered tasks, project-based inquiry, and collaborative interaction are incorporated to support subsequent teaching implementation and process-oriented assessment. Jupyter Notebook functions not only as a digital handout and code execution environment, but also as the activity space connecting these teaching components. The overall framework is shown in **Figure 1**.

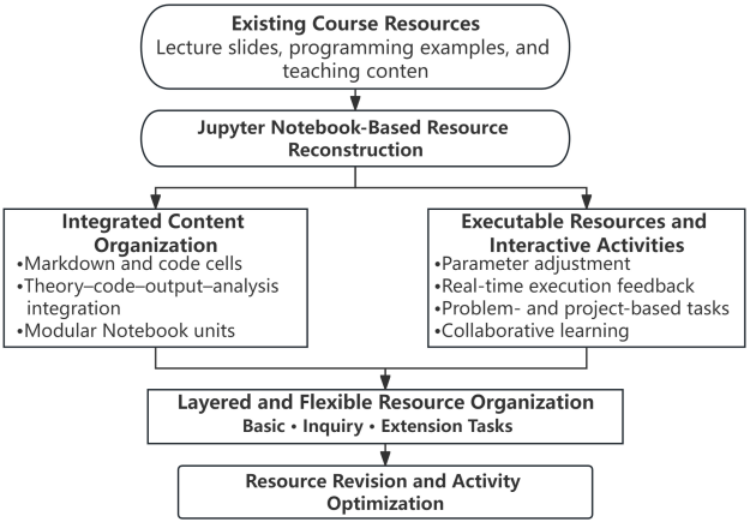

Figure 1. Framework for Jupyter-based course resource reconstruction.

3.1. Integrated organization of teaching content

Each teaching unit consists of Markdown cells and code cells. Markdown cells present conceptual

explanations, mathematical expressions, model structures, task requirements, and operational instructions. The corresponding code immediately follows the theoretical content, and execution results are displayed directly below the relevant code cells, forming a continuous sequence of "principle explanation–code implementation–execution output–result analysis." Instructionally designed Notebooks also emphasize the logical relationships among problem context, model construction, and result interpretation, rather than simply compiling static materials [8].

Different types of course content are organized according to their characteristics; for foundational topics such as tensor operations, activation functions, loss functions, and convolution operations, definitions and computational relationships are introduced first, followed by numerical computation or visualization through code. For more integrated topics such as neural network construction and model training, complete programs are decomposed into data preparation, model definition, parameter configuration, training, testing, and result analysis. This stepwise structure allows students to observe data shapes, network architecture, changes in loss, and prediction results, and to understand the role of each program component within a complete model.

Building on this knowledge structure, integrated teaching units are organized around clearly defined problems or project tasks. Basic tasks require students to execute code, inspect outputs, and explain concepts. Inquiry tasks require them to modify parameters or experimental conditions and compare alternative solutions. Extension tasks guide students toward more complete model construction and result analysis. The resources progress logically from foundational knowledge to model construction and application tasks, while Notebook units can be added, removed, or recombined flexibly according to teaching progress and students' prior knowledge.

3.2. Executable resources and interactive teaching activities

Executable resources present not only final results but also intermediate information such as data shapes, network architecture, parameter counts, training loss, and prediction outputs. Parameters including the learning rate, number of network layers, kernel size, batch size, and number of training epochs are placed in the relevant code cells. After students modify and rerun the code, they can compare changes in model structure and the training process. Thus, interaction in this study primarily refers to students' active adjustment of code and experimental conditions and the direct feedback provided by the Notebook, rather than only to interface operations involving graphical buttons or sliders.

Teaching activities are organized around problem- or project-based tasks. Students first complete basic code execution and result checks independently, and then conduct inquiry by modifying parameters or model structures. When multiple alternatives need to be compared, group members can divide the work, test different configurations, share experimental records in their Notebooks, and jointly explain differences in the results. The instructor provides guidance based on the code, outputs, and written analysis, helping students move from "the program runs" to "the model behavior can be explained." This organization preserves individual operational processes while also providing evidence for collaborative discussion and presentation of outcomes.

To accommodate differences in programming background and learning pace, the course resources are organized into basic, inquiry, and extension levels. Basic tasks ensure mastery of core concepts and operations; inquiry tasks provide room for selecting parameters or model structures; and extension tasks serve students who wish to pursue more advanced learning. Students can rerun key code at their own pace, while instructors provide differentiated feedback based on task completion and the quality of result interpretation.

Because deep learning cases may involve data downloads, pretrained weights, and substantial

computational costs, the resources include instructions on dependencies, data paths, and runtime requirements. Lightweight demonstrations are separated from complete training procedures, and optional processes are controlled through execution switches. Previous research has shown that carefully designed Jupyter resources can organize coherent sequences of conceptual explanation and computational activity while preserving the basic structure of an existing course [9]. The programs are also designed to run in both CPU and GPU environments, reducing the influence of hardware and data conditions on the core teaching content. Through this reconstruction, dispersed lecture slides, programming examples, and teaching content are transformed into coherent, executable, and reusable Notebook units. Tasks at different levels can be combined flexibly according to teaching progress and students' prior knowledge, thereby providing a resource foundation for the representative teaching unit presented in the next section.

4. Design of a representative teaching unit: Convolutional neural networks

Convolutional neural networks cover convolution, pooling, parameter sharing, network construction, model training, and overfitting analysis, forming a relatively complete sequence of concepts and practices. The development of practical cases for Deep Learning courses also emphasizes hierarchical progression and completeness [10]. In this unit, "constructing a handwritten-digit recognition model" serves as the integrated task, and the content is organized through the sequence "foundational computation–code verification–network construction–model training–result analysis."

4.1. Reorganization of teaching content

Conventional slide presentations generally introduce convolution formulas, schematic diagrams, code snippets, and case results as separate knowledge points, leaving students to establish the relationships among theoretical calculations, code definitions, and model execution on their own. For example, output-size calculations, nn.Conv2d parameter settings, and changes in tensor shapes within a network correspond to mathematical representation, program implementation, and model execution, respectively. Without a continuous demonstration, students may find it difficult to understand how these three levels correspond to one another.

In the Notebook, the unit first uses numerical matrices to explain single-channel convolution, making the relationships among the convolution window, kernel, and output elements explicit. It then introduces stride, padding, input and output channels, and parameter counts, and verifies the relevant calculations through PyTorch code. Once the basic operations have been mastered, students proceed to the MNIST handwritten-digit recognition task, which includes data loading, network definition, tensor-shape inspection, model training, and testing. Pooling, ReLU, parameter sharing, and receptive fields are embedded in the corresponding code demonstrations or model-analysis activities, thereby establishing a continuous relationship between foundational principles and the integrated task.

Teaching activities follow the sequence "instructor guidance–student operation–result comparison–explanation and reflection." The instructor uses small-scale numerical examples to explain fundamental principles, after which students execute and modify the code. When they move to the complete model, students are expected to use the understanding developed in earlier activities to explain changes in tensor shapes and parameter settings, rather than simply copying a complete training program.

4.2. Convolution principles and code verification

The Notebook first displays the input matrix, convolution kernel, local window, element-wise products, and summation result, allowing each element in the output feature map to be traced to a specific computational process. Mathematical explanations, input data, program statements, and computational outputs are presented in direct correspondence on the same page.

When learning stride and padding, students first predict the output shape from the input size, kernel size, stride, and padding, and then run the code to verify their calculations. If the theoretical calculation does not match the program output, they return to the parameter definitions and computational process to identify the source of the discrepancy. The parameter count of a convolutional layer is taught similarly: the composition of parameters in single-channel and multi-channel convolutional layers is first explained, after which the shapes of the weight and bias tensors produced by nn.Conv2d are examined to verify the theoretical result.

Inquiry tasks require students to modify the kernel size, stride, padding scheme, or number of output channels and record the resulting changes in output size and parameter count. Basic tasks provide specified parameter combinations, whereas extension tasks require students to design and compare at least two configurations and explain the structural differences. In this way, static formulas are transformed into executable, modifiable, and verifiable computational processes.

4.3. Stepwise organization of the image classification case

The MNIST classification program is decomposed into data loading, sample visualization, network definition, tensor-shape inspection, parameter statistics, single-mini-batch training, and complete training and testing. After data loading, selected images and their labels are displayed so that the input dimensions, number of channels, and class format can be observed directly. After the model is defined, a simulated input is passed through the network layer by layer and the tensor shape is displayed at each stage. This enables students to observe changes in spatial dimensions, feature channels, and class outputs, while identifying common problems such as dimensional mismatches between convolutional and fully connected layers.

The Notebook uses parameter-statistics and model-summary tools to display the number of parameters in each layer and in the complete model, enabling students to analyze the architecture in terms of both data transformations and model scale. To prevent complete training from consuming excessive class time, the unit first uses a single mini-batch to demonstrate the forward pass, loss computation, backpropagation, and parameter updating, while an execution switch controls complete training. The instructor can select the processes to run in class according to the available time and hardware, and students can complete the full training process after class.

During individual inquiry, each student modifies one factor, such as the learning rate, number of output channels, batch size, or number of training epochs, and records the resulting model structure, loss trajectory, and test performance. Students with stronger foundations may further adjust the number of layers or regularization settings. During group collaboration, members test different parameter combinations, record the settings, results, and preliminary interpretations in their Notebooks, and then jointly analyze how the changes affect tensor shapes, parameter counts, the training process, and test performance.

The instructor provides guidance based on the code execution process, intermediate outputs, and quality of analysis. For problems involving dimensional errors, incorrect data paths, or inappropriate training settings, students are guided to use intermediate results to locate the source of the problem. When the code runs successfully, but the explanation is insufficient, the instructor asks students to articulate the relationship

between parameter changes and model behavior, encouraging a shift from program reproduction to conceptual understanding. The treatment of overfitting links training and test performance with regularization, Dropout, and data augmentation, thereby establishing a continuous sequence from model training to problem diagnosis and improvement.

This unit integrates verification of convolution principles, model construction, parameter inquiry, and collaborative analysis into a complete task, demonstrating a design that combines project-based learning, self-directed inquiry, immediate feedback, differentiated learning, and collaborative interaction.

5. Teaching evaluation design and conclusion

Subsequent teaching evaluation can be organized across five dimensions: knowledge mastery, practical competence, learning process, collaborative performance, and learning experience. Knowledge mastery can be assessed through theoretical tests, conceptual explanations, and analysis of model structures. Practical competence can be evaluated through Notebook tasks, code debugging, model construction, and result analysis. The learning process can be examined through task completion, records of code and outputs, and staged assignments. Collaborative performance can be assessed on the basis of task allocation, result sharing, and joint explanation. Learning experience can be investigated through questionnaires, self-assessment, and interviews addressing learning motivation, content difficulty, learning pace, and feedback needs. Jupyter Notebook can also be used to organize formative assessment, preserve learning artifacts, and provide feedback [11].

During subsequent implementation, the Notebook can support pre-class, in-class, and post-class learning. Before class, it can be used for environment checks, review of foundational concepts, and execution of example code. In class, it can support explanation of principles, code verification, project-based inquiry, and group discussion. After class, it can be used for complete training, extension tasks, and result summaries. Evaluation evidence can include theoretical tests, Notebook tasks, project outcomes, learning records, and feedback from instructors and students. Because a full classroom-based empirical study has not yet been conducted, this paper does not assign indicator weights without supporting data and does not make conclusive claims about teaching effectiveness.

Evaluation results and feedback from instructors and students will be used to identify problems in conceptual understanding, code execution, and the runtime environment. Explanatory content, code decomposition, and task difficulty can then be adjusted accordingly, forming a continuous improvement cycle of "teaching implementation–process evaluation–feedback analysis–resource revision."

This paper reconstructs the resources of the Deep Learning course using Jupyter Notebook and integrates project tasks, self-directed inquiry, collaborative interaction, tiered learning, and multidimensional assessment into an executable teaching process. The convolutional neural network unit illustrates the specific organization of theoretical explanation, code verification, model training, and result analysis. Future classroom implementation will be used to refine the resources and activity design further. The approach may also provide a reference for machine learning, data analytics, and other courses that combine theoretical derivation with computational practice.

Funding

Teaching Reform and Research Project of the Yunnan Provincial Higher Education Computer Teaching

Research Association, “Research and Practice on a Jupyter-Based Human–Computer Interactive Teaching Model: A Case Study of the Deep Learning Course” (Project No.: 202403)

Disclosure statement

The authors declare no conflict of interest.

References

[1] Project Jupyter, n.d., Project Jupyter: Interactive Computing. Project Jupyter, visited on July 29, 2026, https://jupyter.org/.

[2] Amoudi G, Tbaishat D, 2023, Interactive Notebooks for Achieving Learning Outcomes in a Graduate Course: A Pedagogical Approach. Education and Information Technologies, 28(12): 16669–16704.

[3] Bascuñana J, León S, González-Miquel M, et al., 2023, Impact of Jupyter Notebook as a Tool to Enhance the Learning Process in Chemical Engineering Modules. Education for Chemical Engineers, 44: 155–163.

[4] Al-Gahmi A, Zhang Y, Valle H, 2022, Jupyter in the Classroom: An Experience Report. Proceedings of the 53rd ACM Technical Symposium on Computer Science Education, 1: 425–431.

[5] Ren Y, Feng Q, Jiang Z, 2022, Exploration and Practice of Jupyter Notebook in Artificial Intelligence Online Teaching. Proceedings of the 2022 International Conference on Educational Innovation and Multimedia Technology: 672–681.

[6] Wei X, Xiao L, 2024, Reform of Integrated Theory–Practice Curriculum and Textbook Construction for Deep Learning under the Background of Emerging Engineering Education. Computer Education, 2024(4): 135–138 + 143.

[7] Yang Y, He G, Liu L, et al., 2022, Teaching Reform of University Deep Learning Courses under an Industry Application Background. Computer Education, 2022(10): 26–30.

[8] Fleischer Y, Biehler R, Schulte C, 2022, Teaching and Learning Data-Driven Machine Learning with Educationally Designed Jupyter Notebooks. Statistics Education Research Journal, 21(2): 7.

[9] Tufino E, Oss S, Alemani M, 2024, Integrating Python Data Analysis in an Existing Introductory Laboratory Course. European Journal of Physics, 45(4): 045707.

[10] Cao J, Li C, Sun X, et al., 2024, Construction of a Practical Teaching Case Library for Deep Learning Courses. Computer Education, 2024(7): 124–128.

[11] Yavuz Temel G, Barenthien J, Padubrin T, 2025, Using Jupyter Notebooks as Digital Assessment Tools: An Empirical Examination of Student Teachers’ Attitudes and Skills Towards Digital Assessment. Education and Information Technologies, 30: 18621–18650.