PetalNet: A Feedforward Neural Network for Classifying Iris Species

Roman Rendon

May 13, 2025

GitHub Repository:
`https://github.com/Romantll/PetalNet`

Abstract

This project presents PetalNet, a fully connected feedforward neural network designed to classify flowers from the Iris dataset. Built from scratch using NumPy, PetalNet achieves high classification accuracy through a 3-hidden-layer architecture. The project includes mathematical derivations of forward propagation and backpropagation, as well as visualizations and training results.

1 Introduction

To understand what this neural network is doing, we must first define the Iris dataset. Originally introduced by British statistician and biologist Ronald Fisher, the Iris dataset contains 150 samples of iris flowers from three distinct species: Setosa, Versicolor, and Virginica. Each sample is described by four numerical features, the length of the sepal, the width of the sepal, the length of the petal, and the width of the petal, which serve as the basis for classification.

To classify this data, we use feedforward neural network, a model inspired by the structure of the human brain. Neural networks consist of layers of interconnected nodes(neurons), where each node processes inputs, applies an activation function, and passes the result to each node in the next layer. In *PetalNet*, the goal is to train a feedforward neural network to accurately map the four input features of each flower to its correct species label.

While there are powerful libraries such as PyTorch's torch.nn module that simplify neural network construction, our goal is to build the network from scratch in Python to understand the underlying mathematics. By manually implementing each component, we gain a deeper understanding of how neural networks process information. In addition to the implementation, we will perform and present selected forward-pass and backpropagation calculations by hand to demonstrate the mathematical foundations of the model. Furthermore, the network architecture is designed with scalability in mind to support more complex datasets in future work.

2 Network Architecture

For the *PetalNet* architecture, we use an input layer of four nodes, each fully connected to the first hidden layer h_1, which contains eight neurons. The network includes three hidden layers in total, with sizes decreasing in depth: $h_1(8), h_2(6)$, and $h_3(3)$. Including multiple hidden layers enables the network to interact with the data in more complex ways. While this level of depth is not necessary for classifying the Iris dataset, it reflects our intent to scale the model for more complex classification tasks and demonstrates the mathematical structure of deep learning models.

The *PetalNet* architecture is structured as follows:

$$\text{Input layer } (4) \to h_1\ (8) \to h_2\ (6) \to h_3\ (3) \to \text{Softmax Output } (3)$$

Next, we define the activation function used in the *PetalNet* architecture. Choosing an appropriate activation function is an important design decision when building a neural

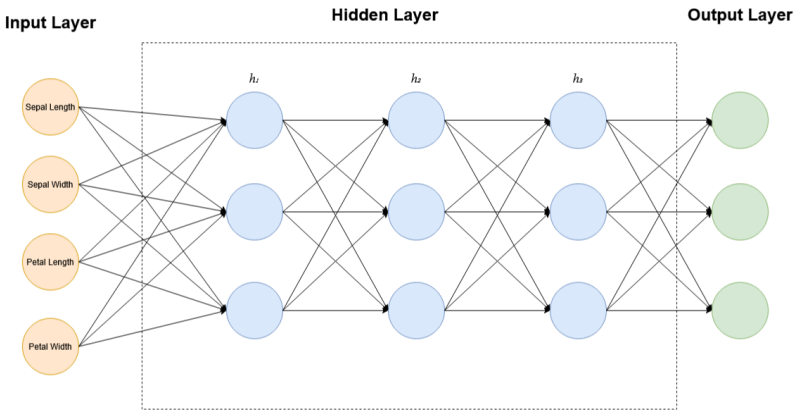

Figure 1: PetalNet architecture: Each layer is fully connected. Only a subset of nodes is shown for clarity.

network, as it determines whether a neuron should be activated and introduces non-linearity into the model. Without this non-linearity, the network would behave like a simple linear regression model regardless of the number of layers, limiting its ability to model complex patterns.

Each hidden layer (h) will utilize the Rectified Linear Unit (ReLU) activation function, defined as:

$$\text{ReLU}(z) = \max(0, z)$$

ReLU is computationally efficient and avoids the vanishing gradient problem associated with other activation functions such as sigmoid or tanh. It outputs zero for negative input values and returns the input itself for positive values. Unlike sigmoid, which relies on exponential functions, ReLU introduces sparsity in the network, often leading to faster and more efficient training. Its simplicity and effectiveness are key reasons for choosing it in this project. Additionally, using ReLU prepares *PetalNet* for future scalability, as the issue of vanishing gradients typically becomes more pronounced in deeper architectures.

We must also define the activation function used in the output layer. For *PetalNet*, we use the softmax function, defined as:

$$\text{softmax}(a)_j = \frac{e^{a_j}}{\sum_{k=1}^{3} e^{a_k}}$$

The softmax function takes the raw output vector from the final hidden layer h_3 and transforms it into a probability distribution over the three output classes. Each element of the

resulting vector lies in the range (0,1), and the sum of all elements equals 1:

$$\sum_{j=1}^{3} \text{softmax}(a_j) = 1$$

This property makes softmax well-suited for multiclass classification tasks, as it allows the output to be interpreted as a vector of class probabilities.

To utilize the probabilities vector, the output layer consists of three neurons, corresponding to the three possible iris species. To match this structure, the target labels are one-hot encoded—each class label is converted into a binary vector where only the position corresponding to the correct class is set to 1. This encoding allows the network's softmax output to be directly compared to the true labels during training using categorical cross-entropy loss.

Finally, the network's output is compared to the one-hot encoded class labels using the categorical cross-entropy loss function, which is well-suited for multi-class classification tasks. This loss penalizes incorrect predictions strongly and works effectively in combination with the softmax activation in the output layer.

3 Forward Pass

In this section, we perform a full forward pass to demonstrate how numerical values interact within the network. While the Iris dataset contains real-valued measurements such as 5.1 cm and 3.5 cm, we use integer-valued inputs and weights in this example for clarity. In practice, neural networks operate on floating-point values, which result in decimal-weighted sums and activations.

To simplify computation, we represent each hidden layer with 3 neurons, such that:

$$h_1(3) \to h_2(3) \to h_3(3) \to \text{Softmax Output}(3)$$

We begin by defining the input vector $x \in \mathbb{R}^4$ as:

$$x = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$$

Next, we define the arbitrary weight matrix $W^{[1]} \in \mathbb{R}^{3\times 4}$ and bias vector $b^{[1]} \in \mathbb{R}^3$ for the first hidden layer:

$$W^{[1]} = \begin{bmatrix} 1 & 0 & -1 & 2 \\ 0 & 1 & 1 & -1 \\ 2 & -1 & 0 & 1 \end{bmatrix}, \quad b^{[1]} = \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix}$$

Computing the first hidden layer using the formula $z^{[1]} = W^{[1]}x + b^{[1]}$, we get:

Neuron 1 in h_1:

$$z_1^{[1]} = w_1^{[1]} \cdot x + b_1^{[1]} = 7$$

so

$$a_1^{[1]} = \text{ReLU}(7) = 7$$

We repeat this process for each neuron in h_1 to form the activation vector that will be mapped to the next layer h_2.

Neuron 2 in h_1:

$$z_2^{[1]} = w_2^{[1]} \cdot x + b_2^{[1]} = 1 \quad \Rightarrow \quad a_2^{[1]} = \text{ReLU}(1) = 1$$

Neuron 3 in h_1:

$$z_3^{[1]} = w_3^{[1]} \cdot x + b_3^{[1]} = 3 \quad \Rightarrow \quad a_3^{[1]} = \text{ReLU}(3) = 3$$

The resulting activation vector of $a^{[1]}$:

$$\begin{bmatrix} 7 \\ 1 \\ 3 \end{bmatrix}$$

We apply the same process to compute the activations for the subsequent hidden layers. Using the activations from h_1 as input, we define the weight matrix $W^{[2]} \in \mathbb{R}^{3\times3}$ and bias vector $b^{[2]} \in \mathbb{R}^3$ for the second hidden layer:

$$W^{[2]} = \begin{bmatrix} 1 & -1 & 2 \\ 0 & 1 & 1 \\ -1 & 0 & 1 \end{bmatrix}, \quad b^{[2]} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

Then:

$$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}, \quad a^{[2]} = \text{ReLU}(z^{[2]})$$

$z^{[2]}$ pre-activation vector:

$$z^{[2]} = \begin{bmatrix} 12 \\ 5 \\ -4 \end{bmatrix} \quad \Rightarrow \quad a^{[2]} = \begin{bmatrix} \text{ReLU}(12) \\ \text{ReLU}(5) \\ \text{ReLU}(-4) \end{bmatrix} = \begin{bmatrix} 12 \\ 5 \\ 0 \end{bmatrix}$$

We repeat this process for h_3:

$$W^{[3]} = \begin{bmatrix} 1 & 1 & 0 \\ -1 & 2 & 1 \\ 0 & -1 & 1 \end{bmatrix}, \quad b^{[3]} = \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix}$$

$z^{[3]}$ pre-activation vector:

$$z^{[3]} = \begin{bmatrix} 18 \\ -2 \\ -6 \end{bmatrix} \quad \Rightarrow \quad a^{[3]} = \begin{bmatrix} \text{R}eLU(18) \\ \text{R}eLU(-2) \\ \text{R}eLU(-6) \end{bmatrix} = \begin{bmatrix} 18 \\ 0 \\ 0 \end{bmatrix}$$

After finally completing computation for each hidden layer this produces the activation vector $a^{[3]} = [18, 0, 0]$, which is then passed to the output layer and transformed into a probability distribution using the softmax function.

For a vector $a = [a_1, a_2, a_3]$ the softmax function is defined:

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

Exponentiating each value:

$$e^{18} \approx 6.05 \times 10^7, \quad e^0 = 1, \quad e^0 = 1$$

Computing the denominator:

$$\sum_{j=1}^{3} e^{a_j} = e^{18} + e^0 + e^0 \approx 6.05 \times 10^7 + 1 + 1 = 6.05 \times 10^7 + 2$$

Calculating the softmax outputs:

$$\hat{y}_1 = \frac{e^{18}}{e^{18} + 2} \approx \frac{6.05 \times 10^7}{6.05 \times 10^7 + 2} \approx 0.99999997$$

$$\hat{y}_2 = \hat{y}_3 = \frac{1}{6.05 \times 10^7 + 2} \approx 1.65 \times 10^{-8}$$

Final softmax output:

$$\hat{y} \approx \begin{bmatrix} 0.99999997 \\ 0.00000002 \\ 0.00000002 \end{bmatrix}$$

This resulting probability distribution indicates that the model is highly confident in classifying the input as belonging to class 1.

As demonstrated, performing these calculations by hand can be a tedious process. This becomes especially true as neural networks grow larger. For example, large language models (LLMs) like ChatGPT contains billions, of parameters. In such models, manually visualizing the underlying computations is impractical. However, understanding the mathematics behind these operations remains crucial. It allows us to modularize neural networks conceptually and appreciate the vast amount of computing power required for them to function effectively.

4 Backpropagation

To demonstrate backpropagation, we compute the gradient of the loss function with respect to a single outer-layer weight, specifically $W_{11}^{[4]}$, which connects the first neuron of the final hidden layer h_3 to the first output neuron corresponding to class 1 (Setosa). We use the forward pass values computed in the previous section.

Step 1: Define relevant values

From the forward pass:

- The activation vector from the last hidden layer:

$$a^{[3]} = \begin{bmatrix} 18 \\ 0 \\ 0 \end{bmatrix}$$

- The softmax output (model prediction):

$$\hat{y} = \begin{bmatrix} 0.99999997 \\ 1.65 \times 10^{-8} \\ 1.65 \times 10^{-8} \end{bmatrix}$$

- The true one-hot label:

$$y = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

Step 2: Apply the chain rule

We compute the gradient using:

$$\frac{\partial \mathcal{L}}{\partial W_{11}^{[4]}} = \frac{\partial \mathcal{L}}{\partial z_1^{[4]}} \cdot \frac{\partial z_1^{[4]}}{\partial W_{11}^{[4]}}$$

- First, since we are using the softmax activation with categorical cross-entropy loss, the derivative simplifies to:

$$\frac{\partial \mathcal{L}}{\partial z_1^{[4]}} = \hat{y}_1 - y_1 = 0.99999997 - 1 = -3 \times 10^{-8}$$

- Then, the partial derivative of $z_1^{[4]}$ with respect to $W_{11}^{[4]}$ is:

$$\frac{\partial z_1^{[4]}}{\partial W_{11}^{[4]}} = a_1^{[3]} = 18$$

Step 3: Compute the gradient

$$\frac{\partial \mathcal{L}}{\partial W_{11}^{[4]}} = (-3 \times 10^{-8}) \cdot 18 = -5.4 \times 10^{-7}$$

Step 4: Gradient descent update

If we use a learning rate $\eta = 0.01$, the weight update becomes:

$$W_{11}^{[4]} \leftarrow W_{11}^{[4]} - \eta \cdot \frac{\partial \mathcal{L}}{\partial W_{11}^{[4]}} = W_{11}^{[4]} + 5.4 \times 10^{-9}$$

This shows that the gradient tells us how to adjust this specific weight to reduce the loss. We use the categorical cross-entropy loss, defined as:

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

where y is the one-hot true label and $\hat{y}$ is the softmax output.

5 Implementation

The *PetalNet* architecture was implemented from scratch in Python using only `numpy` and `scikit-learn` for preprocessing. The implementation consists of modular functions for each part of the forward and backward pass, training loop, and evaluation.

Model Structure

The network was constructed with four layers:

- Input layer: 4 features (sepal and petal measurements)
- Hidden layers: 3 layers with sizes 8, 6, and 3 respectively
- Output layer: 3 neurons corresponding to the 3 iris classes

Each layer is fully connected. ReLU was used for all hidden layers and softmax was applied to the output layer.

```
def forward_pass(X, params):
    Z1 = X @ params['W1'].T + params['b1'].T
    A1 = relu(Z1)
    Z2 = A1 @ params['W2'].T + params['b2'].T
    A2 = relu(Z2)
    Z3 = A2 @ params['W3'].T + params['b3'].T
    A3 = relu(Z3)
    Z4 = A3 @ params['W4'].T + params['b4'].T
    A4 = softmax(Z4)
    return A4, cache
```

Listing 1: Forward Pass Through All Layers

Weight Initialization

All weights were initialized using He initialization:

$$W^{[l]} \sim \mathcal{N}(0, \frac{2}{n_{\text{in}}})$$

where n_{in} is the number of inputs to the layer. Biases were initialized to zero.

```
def initialize_parameters():
    np.random.seed(0)
    return {
        'W1': np.random.randn(8, 4) * np.sqrt(2 / 4),
        'b1': np.zeros((8, 1)),
        'W2': np.random.randn(6, 8) * np.sqrt(2 / 8),
        'b2': np.zeros((6, 1)),
        'W3': np.random.randn(3, 6) * np.sqrt(2 / 6),
        'b3': np.zeros((3, 1)),
        'W4': np.random.randn(3, 3) * np.sqrt(2 / 3),
        'b4': np.zeros((3, 1))
    }
```

Listing 2: He Initialization for Each Layer

Data Preprocessing

The input features were standardized using scikit-learn's `StandardScaler` to have zero mean and unit variance. The output labels were one-hot encoded to match the softmax + cross-entropy formulation.

```
from sklearn.preprocessing import StandardScaler, OneHotEncoder

scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

encoder = OneHotEncoder(sparse_output=False)
y_encoded = encoder.fit_transform(y.reshape(-1, 1))
```

Listing 3: Standardization and One-Hot Encoding

Training

We implemented a custom training loop that runs for a fixed number of epochs. In each iteration, the network:

1. Performs a forward pass to compute predictions
2. Computes the categorical cross-entropy loss
3. Computes gradients using backpropagation

4. Updates all weights and biases using gradient descent

```python
def train(X_train, y_train, epochs=2000, lr=0.01):
    params = initialize_parameters()
    for epoch in range(epochs):
        y_pred, cache = forward_pass(X_train, params)
        loss = compute_loss(y_train, y_pred)
        grads = backward_pass(y_train, params, cache)
        params = update_parameters(params, grads, lr)
        if epoch % 200 == 0:
            print(f"Epoch {epoch} Loss: {loss:.4f}")
    return params
```

Listing 4: Training Loop

This function ties all components together and trains the model using gradient descent over 2000 epochs.

6 Evaluation

After training *PetalNet*, we evaluated its performance on a held-out test set. The model achieved an accuracy of 96.7%, indicating strong classification performance on the Iris dataset.

Test Accuracy

- Final accuracy: 96.7%
- Evaluated on 30 test samples

Sample Predictions (first 10)

```
 1: Predicted=versicolor  Actual=versicolor
 2: Predicted=setosa      Actual=setosa
 3: Predicted=virginica   Actual=virginica
 4: Predicted=versicolor  Actual=versicolor
 5: Predicted=versicolor  Actual=versicolor
 6: Predicted=setosa      Actual=setosa
 7: Predicted=versicolor  Actual=versicolor
 8: Predicted=virginica   Actual=virginica
 9: Predicted=versicolor  Actual=versicolor
10: Predicted=versicolor  Actual=versicolor
```

6.1 Loss Curve

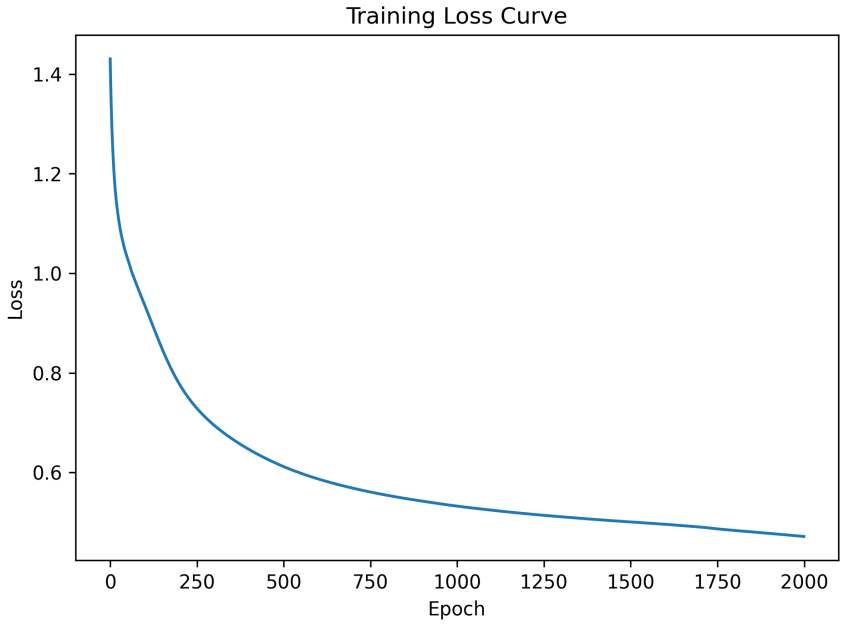

Figure 2: Training Loss over Epochs for PetalNet

Explanation

The model converged quickly, with loss decreasing smoothly as shown in the loss curve. Most misclassifications occurred between Versicolor and Virginica, which are known to have overlapping features in the dataset. The overall performance suggests that the model generalizes well, with no evidence of severe overfitting or underfitting.

7 Conclusion

In this project, we designed and implemented a fully connected feedforward neural network, *PetalNet*, from scratch using Python and `numpy`. The network was trained to classify iris flowers based on four input features, achieving strong accuracy on the classic Iris dataset.

Through a step-by-step breakdown of the forward and backward passes, we demonstrated the mathematical foundations of neural networks, including ReLU activation, softmax output, and backpropagation using the chain rule. We also implemented He initialization and standardized the input features to improve training stability.

PetalNet achieved high accuracy on the test set, indicating successful learning and generalization. However, the network was built using manually defined architecture and learning hyperparameters, and was trained on a relatively small dataset. Future work could explore extending the architecture to handle larger, more complex datasets and experimenting with techniques such as dropout or learning rate schedules.

Overall, this project deepened our understanding of how neural networks work internally, beyond using pre-built libraries. The process of building PetalNet from the ground

up has shown how each component contributes to the network's learning ability and final performance.

8 Future Work

While *PetalNet* successfully classified samples in the Iris dataset with high accuracy, several improvements and extensions are possible.

- **Model Generalization:** Currently, the architecture and hyperparameters (e.g., learning rate, layer sizes) are manually chosen. Future work could involve using grid search or optimization algorithms to automate architecture selection.
- **Regularization:** To improve generalization, techniques such as L2 regularization or dropout could be introduced. This would be especially important when scaling to more complex datasets.
- **Mini-batch Training:** The current implementation uses full-batch gradient descent. Switching to mini-batch training would allow more scalable and efficient training on larger datasets.
- **Model Persistence:** Adding the ability to save and load trained model weights would make the system more flexible and suitable for deployment or experimentation.
- **Transfer to Other Datasets:** PetalNet could be extended to work on other classification tasks beyond the Iris dataset. For example, applying it to higher-dimensional or real-world datasets would test its robustness.
- **Performance Tracking:** Future versions could include more detailed evaluation metrics such as precision, recall, F1 score, and confusion matrices, particularly if the class distribution becomes imbalanced.

These extensions would make PetalNet more robust, extensible, and aligned with modern deep learning workflows.

9 Appendix

9.1 PetalNet Full Source Code

```
import os
import pickle
import numpy as np
from sklearn.datasets import load_iris
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.model_selection import train_test_split

#                         Data Loading & Preprocessing
iris = load_iris()
```

```
X = iris.data
y = iris.target.reshape(-1, 1)

encoder = OneHotEncoder(sparse_output=False)
y_encoded = encoder.fit_transform(y)

X_train, X_test, y_train, y_test = train_test_split(
    X, y_encoded, test_size=0.2, random_state=42
)

scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

#                         Activation Functions
def relu(z): return np.maximum(0, z)
def relu_derivative(z): return (z > 0).astype(float)
def softmax(z):
    exp_z = np.exp(z - np.max(z, axis=1, keepdims=True))
    return exp_z / np.sum(exp_z, axis=1, keepdims=True)

#                         Parameter Initialization
def initialize_parameters():
    np.random.seed(0)
    return {
        'W1': np.random.randn(8, 4) * np.sqrt(2 / 4),
        'b1': np.zeros((8, 1)),
        'W2': np.random.randn(6, 8) * np.sqrt(2 / 8),
        'b2': np.zeros((6, 1)),
        'W3': np.random.randn(3, 6) * np.sqrt(2 / 6),
        'b3': np.zeros((3, 1)),
        'W4': np.random.randn(3, 3) * np.sqrt(2 / 3),
        'b4': np.zeros((3, 1))
    }

#                         Forward Pass
def forward_pass(X, params):
    Z1 = X @ params['W1'].T + params['b1'].T
    A1 = relu(Z1)
    Z2 = A1 @ params['W2'].T + params['b2'].T
    A2 = relu(Z2)
    Z3 = A2 @ params['W3'].T + params['b3'].T
    A3 = relu(Z3)
    Z4 = A3 @ params['W4'].T + params['b4'].T
    A4 = softmax(Z4)
    return A4, {
        'A0': X, 'Z1': Z1, 'A1': A1,
        'Z2': Z2, 'A2': A2,
        'Z3': Z3, 'A3': A3,
        'Z4': Z4, 'A4': A4
    }

#                         Loss
def compute_loss(y_true, y_pred):
```

```
    m = y_true.shape[0]
    log_probs = -np.log(y_pred[range(m), np.argmax(y_true, axis=1)] + 1e
        -9)
    return np.sum(log_probs) / m

#                       Backward Pass
def backward_pass(y_true, params, cache):
    m = y_true.shape[0]
    grads = {}

    dZ4 = (cache['A4'] - y_true) / m
    grads['W4'] = dZ4.T @ cache['A3']
    grads['b4'] = np.sum(dZ4, axis=0, keepdims=True).T

    dA3 = dZ4 @ params['W4']
    dZ3 = dA3 * relu_derivative(cache['Z3'])
    grads['W3'] = dZ3.T @ cache['A2']
    grads['b3'] = np.sum(dZ3, axis=0, keepdims=True).T

    dA2 = dZ3 @ params['W3']
    dZ2 = dA2 * relu_derivative(cache['Z2'])
    grads['W2'] = dZ2.T @ cache['A1']
    grads['b2'] = np.sum(dZ2, axis=0, keepdims=True).T

    dA1 = dZ2 @ params['W2']
    dZ1 = dA1 * relu_derivative(cache['Z1'])
    grads['W1'] = dZ1.T @ cache['A0']
    grads['b1'] = np.sum(dZ1, axis=0, keepdims=True).T

    return grads

#                       Parameter Update
def update_parameters(params, grads, lr):
    for key in params:
        params[key] -= lr * grads[key]
    return params

#                       Training Loop
def train(X_train, y_train, epochs=2000, lr=0.01, print_every=200):
    params = initialize_parameters()
    for epoch in range(1, epochs + 1):
        y_pred, cache = forward_pass(X_train, params)
        loss = compute_loss(y_train, y_pred)
        grads = backward_pass(y_train, params, cache)
        params = update_parameters(params, grads, lr)
        if epoch % print_every == 0:
            print(f"Epoch {epoch:4d}  Loss: {loss:.4f}")
    return params

#                       Evaluation
def evaluate(X, y_true, params):
    y_pred, _ = forward_pass(X, params)
    preds = np.argmax(y_pred, axis=1)
    labels = np.argmax(y_true, axis=1)
```

```
    accuracy = np.mean(preds == labels)
    print(f"\nTest Accuracy: {accuracy*100:.2f}%\n")

    from collections import Counter
    print("Predicted class distribution:", Counter(preds))
    names = load_iris().target_names
    print("\nSample predictions (first 10):")
    for i in range(10):
        print(f" {i+1:2d}: Predicted={names[preds[i]]:<10s}  Actual={names
            [labels[i]]}")

#                           Main
if __name__ == "__main__":
    params = train(X_train, y_train, epochs=2000, lr=0.01, print_every
        =200)
    evaluate(X_test, y_test, params)
```

Listing 5: Complete PetalNet Neural Network Code

References

[1] Scikit-learn. (2024). *Plot Iris dataset*. Retrieved May 14, 2025, from `https://scikit-learn.org/1.4/auto_examples/datasets/plot_iris_dataset.html`

[2] Hidden Layer Neural Network. (2025). Coursera. Retrieved May 14, 2025, from `https://www.coursera.org/articles/hidden-layer-neural-network`

[3] Stanford University. (n.d.). Softmax and logistic regression. In *CS231n: Convolutional Neural Networks for Visual Recognition*. Retrieved May 14, 2025, from `https://cs231n.github.io/linear-classify/#softmax`

[4] Built In. (n.d.). ReLU activation function. Retrieved May 14, 2025, from `https://builtin.com/machine-learning/relu-activation-function`

[5] Stanford University. (n.d.). *Gradient notes* [PDF]. Retrieved May 14, 2025, from `https://web.stanford.edu/class/cs224n/readings/gradient-notes.pdf`

[6] Goel, S. (2017, October 24). Kaiming He initialization. *Medium*. Retrieved May 14, 2025, from `https://medium.com/@shauryagoel/kaiming-he-initialization-a8d9ed0b5899`

[7] Sewell, G. (2014). *Computational methods of linear algebra* (3rd ed.). World Scientific Publishing Company.

[8] Deisenroth, M.P., Faisal, A.A., & Ong, C.S. (2020). *Mathematics for machine learning*. Cambridge University Press.