Final Project: Classifying Iris Species with a Neural Network

Introduction and Guidelines

In this project, you will build and train a feedforward neural network to classify flowers from the Iris dataset. This dataset contains 150 examples of iris flowers, each with four measured features (sepal length, sepal width, petal length, petal width) and one of three possible species labels (Setosa, Versicolor, Virginica).

You will:

- Sketch your proposed network architecture.
- Perform one full forward pass by hand, including activation values and outputs.
- Compute the gradient of the loss with respect to one of the weights in the final layer.
- Implement and train your network using NumPy only (no external ML libraries).

Your network must contain at least one hidden layer. You may use either the sigmoid activation function,

$$\sigma(z) = \frac{1}{1+e^{-z}},$$

or the ReLU activation function,

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

in the hidden layer. The output layer must use the softmax function, which converts raw scores into class probabilities:

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

Because this is a multi-class classification problem, you must use the categorical cross-entropy loss function. This loss compares the output probabilities to the true one-hot encoded class label:

$$E = -\sum_{j=1}^{3} y_j \log(\hat{y}_j),$$

where $y_j \in \{0, 1\}$ indicates the true class label, and $\hat{y}_j$ is the predicted probability of class j. This function penalizes incorrect predictions more sharply than mean squared error and leads to better gradient behavior during training.

The Iris dataset will be provided to you in code using NumPy. You will be responsible for building the network structure, writing forward and backward passes, and performing training using gradient descent. Additional instructions for training and evaluation will follow in the next section.

Understanding One-Hot Encoding

The labels in the Iris dataset are categorical: each flower is either Setosa, Versicolor, or Virginica. Neural networks, however, operate on numerical data. To compute loss using the categorical cross-entropy function, we must convert these string labels into numerical form.

One common method is **one-hot encoding**. This technique represents each class label as a vector of zeros with a single 1 in the position corresponding to the class. For example:

Setosa	$\rightarrow$	$[1, 0, 0]$
Versicolor	$\rightarrow$	$[0, 1, 0]$
Virginica	$\rightarrow$	$[0, 0, 1]$

These vectors are compatible with the `softmax` output layer, which outputs a probability vector of the same shape. The loss function expects the predicted output $\hat{\mathbf{y}} \in \mathbb{R}^3$ to be compared with a true one-hot encoded vector $\mathbf{y} \in \mathbb{R}^3$, so one-hot encoding is required.

This transformation is done using the following Python code:

```
from sklearn.datasets import load_iris
from sklearn.preprocessing import OneHotEncoder

# Load Iris dataset
data = load_iris()
X = data.data
y = data.target.reshape(-1, 1)  # Ensure y is column vector for encoder

# One-hot encode the labels
encoder = OneHotEncoder(sparse=False)
y_encoded = encoder.fit_transform(y)

# Output of y_encoded
print(y_encoded[:5])  # Show first 5 encoded labels
```

The result, `y_encoded`, is a NumPy array of shape (150, 3), where each row is the one-hot representation of the corresponding class label. For instance, for the first five samples:

```
[[1. 0. 0.]  # Setosa
 [0. 1. 0.]  # Versicolor
 [0. 0. 1.]  # Virginica
 [1. 0. 0.]  # Setosa
 [0. 1. 0.]] # Versicolor
```

One can accomplish the same task in matlab with

```
% Load Iris dataset
load fisheriris
% Extract the features and the target labels
X = meas; % Features
y = species; % Labels

% Convert categorical labels to numeric (1, 2, 3)
y_numeric = grp2idx(y); % Convert to numeric labels

% One-hot encode the labels
% Create a matrix for one-hot encoding
y_encoded = full(ind2vec(y_numeric')'); % Convert the numeric labels to one-hot encoding

% Split the data into training and testing sets (80% train, 20% test)
```

```
cv = cvpartition(size(X, 1), 'HoldOut', 0.2); % 80% for training, 20% for testing
X_train = X(training(cv), :);
X_test = X(test(cv), :);
y_train = y_encoded(training(cv), :);
y_test = y_encoded(test(cv), :);

% Display the sizes of the datasets
disp('Training set size:');
disp(size(X_train));
disp('Testing set size:');
disp(size(X_test));
```

This representation makes it possible for the network to compute gradients for all class outputs and to update weights so that the probability for the correct class increases during training.

Deliverables

Your final project submission should include two portions: a written portion and a coded portion. Both portions are required in your submission.

1. Written Portion

The written portion of your project should include the following:

1. **Network Sketch**: Draw a diagram of your neural network architecture. Clearly label the layers, activation functions, and weights. Show how the input layer, hidden layers, and output layer are connected.
2. **Forward Pass**: Compute a single forward pass through the network. You should show the step-by-step calculations for the activations in each layer, including the input, weighted sums, and activations for the hidden and output layers. Use the values of your network's weights, biases, and inputs to do this computation explicitly.
3. **Backpropagation**: For one of your outer-layer weights, compute the gradient of the loss with respect to that weight. Show all steps of the backpropagation process, using the chain rule where necessary. Ensure that you clearly explain the meaning of each term in the gradient.

This portion of the project should be written in a clear and concise manner, with appropriate mathematical explanations and diagrams. Include all steps of the forward pass and backpropagation computations.

2. Coded Portion

The coded portion of your project should include:

1. **Neural Network Implementation**: Implement the neural network design in Python or Matlab. Include all layers, activation functions, and the forward pass calculation. The network should be implemented from scratch, without relying on deep learning libraries like TensorFlow or PyTorch.

2. **Training**: Implement a basic gradient descent algorithm to train the network. Use the Iris dataset (which will be loaded for you) and split the data into training and testing sets. Include a training loop that updates the weights based on the gradients computed during backpropagation.

3. **Evaluation**: Evaluate the performance of your trained network using accuracy on the test set. Ensure that your code outputs the final predictions.

Include screenshots or code snippets showing the output of the network after training. The code should be well-commented to explain each step of the process, and you should ensure that it runs properly on the provided dataset.

Submission Instructions

Submit a PDF file containing:

1. The complete written portion (network sketch, forward pass, backpropagation).

2. Screenshots or code snippets from your implementation in Python or Matlab, along with the output (e.g., accuracy, loss).

Ensure that your PDF is well-organized and easy to read, with clear explanations and appropriate annotations for the code snippets.

Addendum: Derivative of the Cross-Entropy Loss with Softmax Activation

When using the softmax activation function at the output layer, it is common to pair it with the cross-entropy loss function for classification tasks. Let the output layer's preactivation vector be denoted by $\mathbf{a}^{(2)} \in \mathbb{R}^C$, where C is the number of classes. The softmax function produces the predicted probabilities:

$$\hat{\mathbf{y}} = \text{softmax}(\mathbf{a}^{(2)})_i = \frac{e^{a_i^{(2)}}}{\sum_{j=1}^{C} e^{a_j^{(2)}}}.$$

Let the true labels $\mathbf{y}$ be one-hot encoded, so that $\mathbf{y}_i = 1$ if the example belongs to class i, and zero otherwise. The cross-entropy loss is defined as:

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

Using the chain rule, one can compute the gradient of the loss with respect to the preactivation $\mathbf{a}^{(2)}$ as:

$$\frac{\partial L}{\partial a_i^{(2)}} = \hat{y}_i - y_i.$$

Therefore, the gradient simplifies to:

$$\frac{\partial L}{\partial \mathbf{a}^{(2)}} = \hat{\mathbf{y}} - \mathbf{y}.$$

Note: If training on a mini-batch of size m, the average gradient over the batch is:

$$\frac{1}{m} \sum_{k=1}^{m} \left(\hat{\mathbf{y}}^{(k)} - \mathbf{y}^{(k)} \right).$$