Neural Network Implementation Guide: Iris Dataset

This handout provides a concise guide to implementing and training a two-layer neural network from scratch on the Iris dataset. It includes the essential equations and steps needed for forward propagation, loss computation, backpropagation, and parameter updates.

Prerequisites

- Python programming (loops, functions, NumPy).
- Basic linear algebra (vectors, matrices, dot products).
- Familiarity with activation functions (ReLU, sigmoid).
- Understanding of softmax and cross-entropy.

Network Architecture

- **Input layer**: dimension = 4 (features).
- **Hidden layer**: choose H neurons (e.g. H = 10) + activation ϕ.
- **Output layer**: 3 neurons + softmax.

Parameters:

$$W_1 \in \mathbb{R}^{H \times 4}, \quad b_1 \in \mathbb{R}^{H}, \quad W_2 \in \mathbb{R}^{3 \times H}, \quad b_2 \in \mathbb{R}^3$$

Forward Pass Equations

1. Layer 1 pre-activation: $a^{(1)} = W_1 x + b_1$
2. Layer 1 activation: $h = \phi(a^{(1)})$
3. Layer 2 pre-activation: $a^{(2)} = W_2 h + b_2$
4. Softmax: $\hat{y}_k = \frac{e^{a_k^{(2)}}}{\sum_j e^{a_j^{(2)}}}$

Loss Function

Cross-Entropy (single example):

$$L = -\sum_{k=1}^{3} y_k \log(\hat{y}_k)$$

Batch Loss: average over examples.

Backpropagation

Backpropagation computes how to adjust weights and biases to reduce the loss. We apply the chain rule to find how each parameter affects the loss.

Key terms:

- $\delta^{(2)}$: Error at the output layer.
- $\delta^{(1)}$: Error propagated into the hidden layer.
- ϕ': Derivative of the activation function.

Step-by-step

1. **Compute output layer error**:
$$\delta^{(2)} = \hat{y} - y$$
This is how much the prediction $\hat{y}$ differs from the true label y. This works because softmax and cross-entropy simplify the derivative nicely.

2. **Compute gradient w.r.t. W_2**:
$$\frac{\partial L}{\partial W_2} = \delta^{(2)} \, h^T$$
This follows from the chain rule: change in loss from a change in weights depends on the error and the hidden layer activation.

3. **Propagate error to hidden layer**:
$$\delta^{(1)} = (W_2^T \delta^{(2)}) \circ \phi'(a^{(1)})$$
Multiply the error with the transpose of W_2, then element-wise multiply with the derivative of the activation function.

4. **Compute gradient w.r.t. W_1**:
$$\frac{\partial L}{\partial W_1} = \delta^{(1)} \, x^T$$
This is the product of hidden layer error and the input vector.

5. **Bias gradients**:
$$\frac{\partial L}{\partial b_2} = \delta^{(2)}, \qquad \frac{\partial L}{\partial b_1} = \delta^{(1)}$$
Bias gradients are simply the respective error terms, since they don't depend on any inputs.

Implementation Steps

1. Initialize W_1, W_2, b_1, b_2 with small random values.
2. Loop for each epoch:
   - Shuffle training data.
   - For each mini-batch:
     (a) Compute forward pass (Section 3).
     (b) Compute loss and accumulate.
     (c) Compute backprop (Section 4).
     (d) Update parameters: $\theta \leftarrow \theta - \eta \, \nabla_\theta L$ (e.g. SGD).
   - Record training loss and accuracy.

3. Evaluate on test set.
4. Plot loss and accuracy curves vs. epochs.

Good luck! Feel free to annotate your code with notes on each step as you implement.

Useful NumPy Functionality Reference

When implementing your neural network, it is recommended (but not required) to rely on matrix operations using the NumPy library. NumPy provides efficient implementations of many elementary functions on matrices and vectors that can significantly improve performance over native Python and allows you to focus on the concepts being implemented. The following NumPy operations and functions may be helpful during implementation. If you are working in C, you may want to consider implementing some of these functions yourself.

- `np.dot(A, B)` — Matrix multiplication (e.g. for computing activations).
- `A.T` — Transpose of a matrix or vector.
- `np.maximum(0, x)` — ReLU activation function.
- `np.exp(x)` — Exponentiation (used in softmax).
- `np.sum(x, axis=1, keepdims=True)` — Useful for summing across rows.
- `np.random.randn(shape)` — Initialize weights with small random values.
- `np.mean()` — Computing average loss over a batch.
- `np.argmax()` — Getting predicted class from softmax output.
- `np.eye(n)` — Create an identity matrix (helpful for one-hot encoding).
- `np.zeros(shape)` / `np.ones(shape)` — Initialize arrays.

Good luck!