Neural Network Code Reference

Reference Example: XOR Neural Network

The following example demonstrates a fully coded neural network in `numpy` designed to solve the classical XOR problem. The network consists of an input layer with 2 inputs, one hidden layer with 2 neurons, and an output layer with a single neuron. The nonlinearity is provided by the sigmoid activation function, applied both in the hidden and output layers. The training data consists of the 4 possible binary input pairs $(0,0), (0,1), (1,0), (1,1)$, and the targets encode the exclusive-or output. The network is trained using the mean squared error loss

$$L = \frac{1}{n} \sum_{i=1}^{n} \left(y^{(i)} - t^{(i)} \right)^2,$$

where $y^{(i)}$ is the network output and $t^{(i)}$ is the target label for input i.

During training, the network performs both forward propagation and backpropagation explicitly. Gradients are computed manually using the chain rule: the gradient of the loss with respect to the output is used to update weights from the output to the hidden layer, and the error is then propagated backward to update the hidden layer weights.

Python Code

```python
# ===============================
# XOR Neural Network with Numpy
# ===============================

import numpy as np

# -------------------------------
# Activation functions and their derivatives
# -------------------------------
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
    return sigmoid(x) * (1 - sigmoid(x))

# -------------------------------
# XOR data
# -------------------------------
# Inputs
X = np.array([
    [0, 0],
    [0, 1],
    [1, 0],
    [1, 1]
```

```python
])

# Targets
T = np.array([
    [0],
    [1],
    [1],
    [0]
])

# ------------------------------
# Initialize weights and biases
# ------------------------------
np.random.seed(0)
W1 = np.random.randn(2, 2)      # weights from input to hidden (2x2)
b1 = np.random.randn(1, 2)      # bias for hidden layer (1x2)
W2 = np.random.randn(2, 1)      # weights from hidden to output (2x1)
b2 = np.random.randn(1, 1)      # bias for output layer (1x1)

# ------------------------------
# Training loop
# ------------------------------
learning_rate = 0.1
epochs = 10000

for epoch in range(epochs):
    # -------- Forward pass --------
    z1 = X @ W1 + b1                # input to hidden layer
    a1 = sigmoid(z1)               # output from hidden layer

    z2 = a1 @ W2 + b2              # input to output layer
    y = sigmoid(z2)                # final prediction

    # -------- Error computation --------
    error = y - T
    loss = np.mean(error**2)

    # -------- Backpropagation --------
    dE_dy = 2 * (y - T)                          # dL/dy
    dy_dz2 = sigmoid_derivative(z2)             # dy/dz2
    dz2_dW2 = a1                                # hidden activations

    dE_dW2 = dz2_dW2.T @ (dE_dy * dy_dz2)      # gradient for W2
    dE_db2 = np.sum(dE_dy * dy_dz2, axis=0, keepdims=True)

    dz2_da1 = W2                                # from output to hidden
    da1_dz1 = sigmoid_derivative(z1)

    dE_dz1 = (dE_dy * dy_dz2) @ dz2_da1.T * da1_dz1
    dE_dW1 = X.T @ dE_dz1                       # gradient for W1
    dE_db1 = np.sum(dE_dz1, axis=0, keepdims=True)

    # -------- Update weights and biases --------
    W2 -= learning_rate * dE_dW2
    b2 -= learning_rate * dE_db2
    W1 -= learning_rate * dE_dW1
    b1 -= learning_rate * dE_db1

    # Print loss every 1000 steps
```

```python
    if epoch % 1000 == 0:
        print(f"Epoch {epoch}, Loss: {loss:.5f}")

# ------------------------------
# Final output
# ------------------------------
print("\nFinal predictions after training:")
print(np.round(y, 3))
```

Takeaways

The script constructs the full pipeline of a feedforward neural network:

- The preactivations a_1 and a_2 are computed using affine maps (weights +biases).
- The activations z_1 and z_2 are computed using sigmoid.
- The gradients of the loss with respect to all parameters are computed explicitly using the chain rule.

A template to get you started on your project

```python
import numpy as np
from sklearn.datasets import load_iris
from sklearn.preprocessing import OneHotEncoder
from sklearn.model_selection import train_test_split

# Load the Iris dataset
data = load_iris()
X = data.data  # shape: (150, 4)
y = data.target.reshape(-1, 1)  # reshape for encoder

# One-hot encode the labels
encoder = OneHotEncoder(sparse_output=False)
y_encoded = encoder.fit_transform(y)  # shape: (150, 3)

# Normalize features (optional)
X = (X - np.mean(X, axis=0)) / np.std(X, axis=0)

# Split into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y_encoded, test_size=0.2, random_state=42
)

# Set network dimensions
input_dim = 4     # number of features
hidden_dim = 6    # size of hidden layer
output_dim = 3    # number of classes
```

```
# Initialize weights and biases


# Activation functions
def relu(a):


def relu_derivative(a):


def softmax(a):


# Loss function: cross-entropy
def cross_entropy(y_pred, y_true):
    eps = 1e-10
    return -np.mean(np.sum(y_true * np.log(y_pred + eps), axis=1))

# Gradient of loss w.r.t. logits (output preactivation)
def dL_da2(y_pred, y_true):


# Training loop
for epoch in range(10000):
       # Forward pass
              # preactivation hidden layer
                       # activation hidden layer
                    # preactivation output layer
                # activation output layer (softmax)

    # Compute loss
    loss = cross_entropy(y_pred, y_train)

    # Backpropagation
    da2 = dL_da2(y_pred, y_train)        # dL/da2
    dW2 = z1.T @ da2                     # dL/dW2
    db2 = np.sum(da2, axis=0, keepdims=True)

    dz1 = da2 @ W2.T                     # dL/dz1
    da1 = dz1 * relu_derivative(a1)      # dL/da1

    dW1 = X_train.T @ da1
    db1 = np.sum(da1, axis=0, keepdims=True)

    # Update parameters
    lr =    #Learning Rate
        #Update W1
```

```
        #Update b1
        #Update W2
        #update b2

    # Print loss every 10 epochs
    if epoch % 100 == 0:
        print(f"Epoch {epoch}, Loss: {loss:.4f}")

# test accuracy evaluation

# Final predictions on test set
a1_test = np.dot(X_test, W1) + b1
z1_test = relu(a1_test)
a2_test = np.dot(z1_test, W2) + b2
z2_test = softmax(a2_test)

# Print predicted vs actual for comparison
print("Final Predicted Output (first 5 samples):")
print(np.round(y_pred[:5], decimals=3))  # Rounded for readability

print("\nTarget Output (first 5 samples):")
print(y_test[:5])
```

And the same program in Matlab:

```
% ================================
% Iris Neural Network in MATLAB
% ================================

% Load the Iris dataset
load fisheriris
X = meas;                          % shape: 150×4
y_labels = grp2idx(species);     % categorical to numeric: 1, 2, 3
y = full(ind2vec(y_labels'))';   % one-hot encode to 150×3

% Normalize features
X = (X - mean(X)) ./ std(X);

% Split into training and testing sets (80/20)
cv = cvpartition(size(X,1), 'HoldOut', 0.2);
X_train = X(training(cv), :);
y_train = y(training(cv), :);
X_test  = X(test(cv), :);
y_test  = y(test(cv), :);

% Set network dimensions
```

```
input_dim = 4;
hidden_dim = 6;
output_dim = 3;

% Initialize weights and biases
rng(0);  % for reproducibility
W1 = randn(input_dim, hidden_dim);
b1 = zeros(1, hidden_dim);
W2 = randn(hidden_dim, output_dim);
b2 = zeros(1, output_dim);

% Activation functions
relu =
relu_derivative =
softmax_fn =  % numerical stability handled via shifting if needed

% Loss function (cross-entropy)
cross_entropy = @(y_pred, y_true) -mean(sum(y_true .* log(y_pred + 1e-10), 2));

% Gradient of cross-entropy w.r.t. logits
dL_da2 =

% Training loop
epochs = 10000;
lr = ;

for epoch = 1:epochs
    % -------- Forward pass --------
           % preactivation
                 % activation
              % preactivation output
          % softmax output

    % -------- Loss computation --------
    loss = cross_entropy(y_pred, y_train);

    % -------- Backpropagation --------
    da2 = dL_da2(y_pred, y_train);
    dW2 = z1' * da2;
    db2 = sum(da2, 1);

    dz1 = da2 * W2';
    da1 = dz1 .* relu_derivative(a1);
    dW1 = X_train' * da1;
    db1 = sum(da1, 1);

    % -------- Update weights and biases --------
    W1 =
```

```
    b1 =
    W2 =
    b2 =

    % Print loss every 100 epochs
    if mod(epoch, 100) == 0
        fprintf('Epoch %d, Loss: %.4f\n', epoch, loss);
    end
end

% -------- Evaluation on test data --------
a1_test = X_test * W1 + b1;
z1_test = relu(a1_test);
a2_test = z1_test * W2 + b2;
z2_test = softmax_fn(a2_test);

% Display final prediction vs. target for first 5 test samples
disp("Final Predicted Output (first 5 samples):")
disp(round(z2_test(1:5, :), 3))

disp("Target Output (first 5 samples):")
disp(y_test(1:5, :))
```