Konchada. Shyam sai RA2311026010305 AF1

Artificial Neural Network – Backpropagation

Code :

```
# Pure NumPy backprop demo (Colab-ready)
import numpy as np
np.random.seed(42)

# ---------- Toy dataset: XOR ----------
X = np.array([[0,0],
              [0,1],
              [1,0],
              [1,1]], dtype=np.float64)   # shape (4,2)
y = np.array([[0],[1],[1],[0]], dtype=np.float64)  # shape (4,1)
n_samples = X.shape[0]

# ---------- Utilities ----------
def sigmoid(z):
    return 1.0 / (1.0 + np.exp(-z))

def sigmoid_grad(a):   # a = sigmoid(z)
    return a * (1.0 - a)

def relu(z):
    return np.maximum(0, z)
```

```
def relu_grad(z):
    return (z > 0).astype(np.float64)


def bce_loss(y_true, y_pred, eps=1e-12):
    return -np.mean(y_true * np.log(y_pred + eps) + (1 - y_true) * np.log(1 - y_pred
+ eps))


# ---------- Initialize parameters ----------
input_dim = 2
hidden_dim = 3
output_dim = 1


W1 = np.random.randn(input_dim, hidden_dim) * 0.5
b1 = np.zeros((1, hidden_dim))
W2 = np.random.randn(hidden_dim, output_dim) * 0.5
b2 = np.zeros((1, output_dim))


# ---------- Forward pass function ----------
def forward(X, W1, b1, W2, b2):
    z1 = X.dot(W1) + b1        # (N, hidden_dim)
    a1 = relu(z1)              # (N, hidden_dim)
    z2 = a1.dot(W2) + b2       # (N, 1)
    a2 = sigmoid(z2)           # (N, 1)
    return {"z1": z1, "a1": a1, "z2": z2, "a2": a2}
```

```
# ---------- Backprop (analytical) ----------
def backprop(X, y, cache, W2):
    N = X.shape[0]
    a1 = cache["a1"]
    a2 = cache["a2"]
    z1 = cache["z1"]

    # For BCE loss with sigmoid output, dL/dz2 = (a2 - y) / N
    dz2 = (a2 - y) / N                          # (N,1)
    dW2 = a1.T.dot(dz2)                         # (hidden_dim,1)
    db2 = np.sum(dz2, axis=0, keepdims=True)   # (1,1)

    da1 = dz2.dot(W2.T)                         # (N, hidden_dim)
    dz1 = da1 * relu_grad(z1)                   # (N, hidden_dim)
    dW1 = X.T.dot(dz1)                          # (input_dim, hidden_dim)
    db1 = np.sum(dz1, axis=0, keepdims=True)    # (1, hidden_dim)

    return {"dW1": dW1, "db1": db1, "dW2": dW2, "db2": db2, "dz2": dz2}

# ---------- Numeric gradient check (finite differences) ----------
def numeric_grad_check(param_matrix, param_name, analytical_grad,
forward_loss_fn, n_checks=5, eps=1e-6):
    print(f"\nNumeric gradient check for {param_name}:")
    # flatten indices to choose random elements to check
```

```
    flat_size = param_matrix.size
    indices = np.random.choice(flat_size, min(n_checks, flat_size), replace=False)
    param_flat = param_matrix.flatten()
    analytic_flat = analytical_grad.flatten()

    for idx in indices:
        orig = param_flat[idx]
        # plus
        param_flat[idx] = orig + eps
        param_plus = param_flat.reshape(param_matrix.shape)
        loss_plus = forward_loss_fn(param_plus, which=param_name)
        # minus
        param_flat[idx] = orig - eps
        param_minus = param_flat.reshape(param_matrix.shape)
        loss_minus = forward_loss_fn(param_minus, which=param_name)
        # restore
        param_flat[idx] = orig

        num_grad = (loss_plus - loss_minus) / (2 * eps)
        ana_grad = analytic_flat[idx]
        rel_err = abs(num_grad - ana_grad) / (abs(num_grad) + abs(ana_grad) + 1e-12)
        print(f" idx {idx}: numeric = {num_grad:.6e}, analytic = {ana_grad:.6e}, rel_err = {rel_err:.3e}")
```

```
# ---------- helper that recomputes loss with a modified param ----------
def loss_with_perturbed_param(modified_param, which):
    # we will replace the named parameter with modified_param and compute loss
    global W1, b1, W2, b2
    if which == "W1":
        cache = forward(X, modified_param, b1, W2, b2)
    elif which == "b1":
        cache = forward(X, W1, modified_param, W2, b2)
    elif which == "W2":
        cache = forward(X, W1, b1, modified_param, b2)
    elif which == "b2":
        cache = forward(X, W1, b1, W2, modified_param)
    else:
        raise ValueError("Unknown param name")
    return bce_loss(y, cache["a2"])

# ---------- Take one forward/backprop step and print ----------
cache = forward(X, W1, b1, W2, b2)
loss_before = bce_loss(y, cache["a2"])
grads = backprop(X, y, cache, W2)

print("=== Forward pass results ===")
print("a2 (predictions):\n", cache["a2"])
print(f"Loss before update: {loss_before:.6f}")
```

```
print("\n=== Analytic gradients (backprop) ===")

print("dW2 shape:", grads["dW2"].shape)

print("dW2:\n", grads["dW2"])

print("db2:\n", grads["db2"])

print("dW1 shape:", grads["dW1"].shape)

print("dW1:\n", grads["dW1"])

print("db1:\n", grads["db1"])


# ---------- Numeric gradient checks on W2 and W1 (random few elements) ----------

numeric_grad_check(W2, "W2", grads["dW2"], loss_with_perturbed_param,
n_checks=5)

numeric_grad_check(W1, "W1", grads["dW1"], loss_with_perturbed_param,
n_checks=5)


# ---------- Apply one gradient descent update to verify loss decreases ----------

lr = 0.5

W1_new = W1 - lr * grads["dW1"]

b1_new = b1 - lr * grads["db1"]

W2_new = W2 - lr * grads["dW2"]

b2_new = b2 - lr * grads["db2"]


cache_after = forward(X, W1_new, b1_new, W2_new, b2_new)

loss_after = bce_loss(y, cache_after["a2"])
```

print(f"\nLoss after one gradient step (lr={lr}): {loss_after:.6f}")

print("Loss decreased?:", loss_after < loss_before)

Output :

```
=== Forward pass results ===
a2 (predictions):
 [[0.5       ]
 [0.645953  ]
 [0.52998549]
 [0.678931  ]]
Loss before update: 0.725295

=== Analytic gradients (backprop) ===
dW2 shape: (3, 1)
dW2:
 [[ 0.07482247]
 [ 0.        ]
 [-0.00295625]]
db2:
 [[0.08871737]]
dW1 shape: (2, 3)
dW1:
 [[ 0.04124045  0.         -0.01226012]
 [ 0.06413262  0.         -0.03984259]]
db1:
 [[-0.028649    0.         -0.01226012]]
```

```
Numeric gradient check for W2:
 idx 1: numeric = 0.000000e+00, analytic = 0.000000e+00, rel_err = 0.000e+00
 idx 2: numeric = -2.956253e-03, analytic = -2.956253e-03, rel_err = 2.237e-09
 idx 0: numeric = 7.482247e-02, analytic = 7.482247e-02, rel_err = 5.440e-10

Numeric gradient check for W1:
 idx 2: numeric = -1.226012e-02, analytic = -1.226012e-02, rel_err = 2.463e-09
 idx 5: numeric = -3.984259e-02, analytic = -3.984259e-02, rel_err = 2.766e-10
 idx 0: numeric = 4.124045e-02, analytic = 4.124045e-02, rel_err = 2.463e-10
 idx 4: numeric = 0.000000e+00, analytic = 0.000000e+00, rel_err = 0.000e+00
 idx 1: numeric = 0.000000e+00, analytic = 0.000000e+00, rel_err = 0.000e+00

Loss after one gradient step (lr=0.5): 0.716380
Loss decreased?: True
```

Inference :

 Inputs go forward through the network layer by layer to produce predictions.

□ The hidden layer uses ReLU, turning negatives into zero and keeping positives.

□ The output layer uses sigmoid, giving probabilities between 0 and 1.

□ Loss (error) tells how far predictions are from actual labels.

□ Backpropagation works backwards, starting from the output layer and moving to earlier layers.

□ It computes gradients, which show how each weight/bias affects the loss.

□ Positive gradient → decrease weight, negative gradient → increase weight.

□ A gradient check confirms the computed gradients are correct.

□ After one weight update step, the loss usually decreases.

□ This cycle — forward, loss, backprop, update — is the basic learning process of all neural networks.