~\Documents\machine learning\code examples\Qwen_python, fc and conv backprop 2 (everthing is a function).py

```python
import numpy as np

# ================== FORWARD PASS FUNCTIONS ==================

def conv2d(image, kernel):
    h, w = image.shape
    kh, kw = kernel.shape
    out_h = h - kh + 1
    out_w = w - kw + 1
    output = np.zeros((out_h, out_w))
    for i in range(out_h):
        for j in range(out_w):
            region = image[i:i+kh, j:j+kw]
            output[i, j] = np.sum(region * kernel)
    return output

def relu(x):
    return np.maximum(0, x)

def max_pooling(x, size=2, stride=2):
    h, w = x.shape
    out_h = h // size
    out_w = w // size
    output = np.zeros((out_h, out_w))
    for i in range(out_h):
        for j in range(out_w):
            region = x[i*stride:i*stride+size, j*stride:j*stride+size]
            output[i, j] = np.max(region)
    return output

def flatten(x):
    return x.flatten()

def fully_connected(x, weights, bias):
    return np.dot(weights, x) + bias

def softmax(x):
    exps = np.exp(x - np.max(x))  # Numerically stable
    return exps / np.sum(exps)

def cross_entropy_loss(probs, label):
    return -np.log(probs[label] + 1e-10)


# ================== BACKWARD PASS FUNCTIONS ==================

def grad_fully_connected(x, weights, probs, label):
    dlogits = probs.copy()
    dlogits[label] -= 1  # derivative of CE loss w.r.t logits
    dfc_weights = np.outer(dlogits, x)
    dfc_bias = dlogits
    dx = np.dot(weights.T, dlogits)
    return dfc_weights, dfc_bias, dx

def unflatten_gradient(flat_grad, shape=(13,13)):
    return flat_grad.reshape(shape)

def grad_max_pool(dpool_out, from_relu_shape, size=2, stride=2):
    d_relu = np.zeros(from_relu_shape)
    ph, pw = dpool_out.shape

    for i in range(ph):
        for j in range(pw):
            region = np.zeros((size, size))
            region_idx = np.unravel_index(np.argmax(region), region.shape)
            region[region_idx] = dpool_out[i,j]
            d_relu[i*stride:i*stride+size, j*stride:j*stride+size] += region
    return d_relu

def grad_relu(d_after_relu, pre_relu):
    d_relu = d_after_relu.copy()
    d_relu[pre_relu <= 0] = 0
    return d_relu

def grad_conv(image, d_conv_out, kernel_shape):
    dkernel = np.zeros(kernel_shape)
    kh, kw = kernel_shape
    dh, dw = d_conv_out.shape

    for i in range(dh):
        for j in range(dw):
            region = image[i:i+kh, j:j+kw]
            dk = region * d_conv_out[i,j]
            dkernel += dk
    return dkernel


# ================== MAIN TRAINING LOOP ==================

# Fake input and label
image = np.random.rand(28, 28)  # fake grayscale image
true_label = 3  # pretend this is class 3

# Initialize filter and FC weights
kernel = np.random.randn(3, 3) * 0.01
fc_weights = np.random.randn(10, 13*13) * 0.01
fc_bias = np.zeros(10)
learning_rate = 0.01

# --- FORWARD PASS ---
conv_out = conv2d(image, kernel)
relu_out = relu(conv_out)
pool_out = max_pooling(relu_out)
flat = flatten(pool_out)
logits = fully_connected(flat, fc_weights, fc_bias)
```

```python
probs = softmax(logits)
loss = cross_entropy_loss(probs, true_label)

print("Initial prediction:", np.argmax(probs))
print("Loss:", loss)

# --- BACKWARD PASS ---
dfc_weights, dfc_bias, dx_flat = grad_fully_connected(flat, fc_weights, probs, true_label)
dx_pool = unflatten_gradient(dx_flat)
dx_relu = grad_max_pool(dx_pool, relu_out.shape)
dx_conv = grad_relu(dx_relu, conv_out)
dkernel = grad_conv(image, dx_conv, kernel.shape)

# --- UPDATE WEIGHTS ---
fc_weights -= learning_rate * dfc_weights
fc_bias -= learning_rate * dfc_bias
kernel -= learning_rate * dkernel

# --- RE-FORWARD PASS TO CHECK IMPROVEMENT ---
conv_out = conv2d(image, kernel)
relu_out = relu(conv_out)
pool_out = max_pooling(relu_out)
flat = flatten(pool_out)
logits = fully_connected(flat, fc_weights, fc_bias)
probs = softmax(logits)
loss = cross_entropy_loss(probs, true_label)

print("\nAfter one update:")
print("Prediction:", np.argmax(probs))
print("Loss:", loss)
```