004. ChatGPT, fc and conv backprop (fixed).py

```python
import numpy as np  # numerical arrays, linear algebra, randoms — our workhorse

# =========================
# Forward-pass primitives
# =========================

def conv2d(image, kernel):
    """
    Naive 2D 'valid' convolution (really cross-correlation — no kernel flip).
    image: 2D array (H x W)
    kernel: 2D array (kH x kW)
    returns: 2D array of shape ((H - kH + 1) x (W - kW + 1))
    """
    h, w = image.shape                        # input spatial dims
    kh, kw = kernel.shape                     # kernel spatial dims
    out_h = h - kh + 1                        # 'valid' conv height
    out_w = w - kw + 1                        # 'valid' conv width
    output = np.zeros((out_h, out_w))         # allocate result buffer

    # slide kernel over every valid top-left position
    for i in range(out_h):                    # row loop over output
        for j in range(out_w):                # col loop over output
            region = image[i:i+kh, j:j+kw] # current receptive field (kH x kW)
            output[i, j] = np.sum(region * kernel)  # elementwise mult + sum
    return output                             # (out_h x out_w)


def relu(x):
    """
    Elementwise ReLU: max(0, x) applied to every entry.
    We use np.maximum (vectorized), not np.max (which would reduce to a scalar).
    """
    return np.maximum(0, x)                   # same shape as x

```

```python

def max_pooling(x, size=2, stride=2):
    """
    2D max pooling.
    x: 2D array after ReLU (H x W)
    size: pooling window (size x size)
    stride: step between windows
    returns: pooled map of shape:
             out_h = floor((H - size)/stride) + 1
             out_w = floor((W - size)/stride) + 1
    """
    h, w = x.shape
    out_h = (h - size) // stride + 1       # generic formula (not assuming size==stride, though default matches)
    out_w = (w - size) // stride + 1
    output = np.zeros((out_h, out_w))      # pooled output

    for i in range(out_h):                 # iterate vertical positions
        for j in range(out_w):             # iterate horizontal positions
            # extract the pooling window using stride for position and size for window extent
            region = x[i*stride:i*stride+size, j*stride:j*stride+size]
            output[i, j] = np.max(region)  # route forward the maximum
    return output


def flatten(x):
    """
    Flatten a 2D feature map to 1D vector (row-major).
    """
    return x.flatten()                      # shape: (H*W,)


def fully_connected(x, weight, bias):
    """
    Dense layer: y = W x + b
    weight: (num_classes x D)
    bias: (num_classes,)
    x: (D,)
```

```
    returns: (num_classes,)
    """
    return np.dot(weight, x) + bias


def softmax(x):
    """
    Numerically-stable softmax.
    x: logits (K,)
    returns: probabilities summing to 1 (K,)
    """
    shifted = x - np.max(x)                 # stabilize exponentials
    exps = np.exp(shifted)                  # exp per class
    return exps / np.sum(exps)              # normalize to simplex


def cross_entropy_loss(probs, label):
    """
    Negative log-likelihood for a single label.
    probs: softmax probs (K,)
    label: integer class id
    """
    return -np.log(probs[label] + 1e-10)    # 1e-10 to avoid log(0) tantrums


# ==================================
# Backward-pass (autodiff by hand)
# ==================================

def grad_fully_connected(x, weights, probs, label):
    """
    Gradients for FC layer when loss = cross-entropy(softmax(logits), label).
    For softmax + CE, dL/dlogits = probs with 1 subtracted at the true class.
    x: input vector to FC (D,)
    weights: (K x D)
    probs: softmax output (K,)
    label: integer
```

```
    returns:
      dW: (K x D), db: (K,), dx: (D,)
    """
    dlogits = probs.copy()                  # do NOT mutate probs in-place
    dlogits[label] -= 1.0                   # softmax-CE gradient magic

    dW = np.outer(dlogits, x)               # each row k: dlogits[k] * x
    db = dlogits                            # derivative wrt bias is just dlogits
    dx = np.dot(weights.T, dlogits)         # push gradient back to input vector
    return dW, db, dx


def unflatten_gradient(flat_grad, shape):
    """
    Reshape a flat gradient vector back to the pooled map shape.
    flat_grad: (H*W,)
    shape: tuple (H,W) to restore
    """
    return flat_grad.reshape(shape)         # exact inverse of flatten()


def grad_max_pool(dpool_out, relu_out, size=2, stride=2):
    """
    Backprop through max-pooling.
    dpool_out: gradient arriving from above, shape (out_h, out_w)
    relu_out: the original input to pooling (post-ReLU map), shape (H,W)
    We re-find argmax in each pooling window and send all gradient to that spot.
    returns: gradient wrt relu_out, shape (H,W)
    """
    d_relu = np.zeros_like(relu_out)        # initialize with zeros (only argmax gets gradient)
    ph, pw = dpool_out.shape                # pooled spatial dims

    for i in range(ph):                     # loop pooled rows
        for j in range(pw):                 # loop pooled cols
            # slice the exact window that produced pooled value
            region = relu_out[i*stride:i*stride+size, j*stride:j*stride+size]
            # index of the maximum inside the region (ties go to the first max)
```

```
            max_pos = np.unravel_index(np.argmax(region), region.shape)
            # route upstream gradient ONLY to that max position
            d_relu[i*stride + max_pos[0], j*stride + max_pos[1]] += dpool_out[i, j]
    return d_relu


def grad_relu(d_after_relu, pre_relu):
    """
    Backprop through ReLU.
    d_after_relu: gradient wrt ReLU output
    pre_relu: the tensor BEFORE ReLU (to know where it was <= 0)
    returns: gradient wrt pre-ReLU input (zero where pre_relu <= 0)
    """
    d = d_after_relu.copy()                 # avoid mutating caller's buffer
    d[pre_relu <= 0] = 0.0                  # gradient blocked where ReLU was off
    return d


def grad_conv(image, d_conv_out, kernel_shape):
    """
    Gradient wrt the kernel for our 'valid' conv.
    image: input (H x W)
    d_conv_out: gradient wrt conv output (H-kH+1 x W-kW+1)
    kernel_shape: (kH, kW)
    returns: dKernel (kH x kW)
    NOTE: We don't compute dImage here (not needed to update weights in this toy).
    """
    dkernel = np.zeros(kernel_shape)        # accumulator for kernel gradient
    kh, kw = kernel_shape                   # kernel dims
    dh, dw = d_conv_out.shape               # output gradient dims

    for i in range(dh):                     # slide over every output location
        for j in range(dw):
            region = image[i:i+kh, j:j+kw]  # input patch that contributed
            dkernel += region * d_conv_out[i, j]  # linearity lets us sum contributions
    return dkernel

```

```python

# =========================
# Tiny training demo (1 step)
# =========================

# Reproducibility — keep the RNG civilized
np.random.seed(42)

# Fake input and label for the demo
image = np.random.rand(28, 28)              # a pretend grayscale image (MNIST-ish)
true_label = 3                               # arbitrarily pick class 3 as the target

# Initialize parameters (small randoms to avoid early saturation)
kernel = np.random.randn(3, 3) * 0.01       # single 3x3 conv filter
# After conv (28->26) and 2x2 pool stride 2 (26->13), we have 13*13 features
fc_in_dim = 13 * 13
num_classes = 10
fc_weights = np.random.randn(num_classes, fc_in_dim) * 0.01  # FC weight matrix
fc_bias = np.zeros(num_classes)             # bias starts at zeros

learning_rate = 0.01                        # da tiny SGD step

# ---------- FORWARD PASS ----------
conv_out = conv2d(image, kernel)            # (26 x 26)
relu_out = relu(conv_out)                    # (26 x 26), threshold at 0
pool_out = max_pooling(relu_out, size=2, stride=2)  # (13 x 13)
flat = flatten(pool_out)                     # (169,)
logits = fully_connected(flat, fc_weights, fc_bias) # (10,)
probs = softmax(logits)                      # (10,), sums to 1
loss = cross_entropy_loss(probs, true_label)# scalar

print("Initial prediction:", np.argmax(probs))  # which class had the max prob
print("Loss:", float(loss))                     # cast to float for prettier print

# ---------- BACKWARD PASS ----------
# Gradients through FC
dfc_W, dfc_b, d_flat = grad_fully_connected(flat, fc_weights, probs, true_label)  # shapes: (10x169), (10,), (169,)
```

```

# Reshape gradient back to pooled map shape
d_pool = unflatten_gradient(d_flat, pool_out.shape)   # (13 x 13)

# Backprop through max-pool (needs the *forward* relu_out to re-find argmax)
d_relu_from_pool = grad_max_pool(d_pool, relu_out, size=2, stride=2)  # (26 x 26)

# Backprop through ReLU to get gradient wrt conv_out (pre-ReLU)
d_conv_out = grad_relu(d_relu_from_pool, conv_out)   # (26 x 26)

# Gradient wrt kernel from conv layer
dkernel = grad_conv(image, d_conv_out, kernel.shape) # (3 x 3)

# ---------- SGD PARAM UPDATE ----------
fc_weights -= learning_rate * dfc_W       # descend on FC weights
fc_bias    -= learning_rate * dfc_b       # descend on FC bias
kernel     -= learning_rate * dkernel     # descend on conv kernel

# ---------- RE-FORWARD (sanity poke) ----------
conv_out = conv2d(image, kernel)          # recompute with updated params
relu_out = relu(conv_out)
pool_out = max_pooling(relu_out, size=2, stride=2)
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:", float(loss))

```