T3.1 : Convolutional neural networks (CNNs)

Forward and backward pass in a CNN binary classifier

Given

(1) A simple CNN having the following architecture:

Layer	Output size	Kernel size	Stride	Padding	Activation function
Input	5x5x1	-	-	-	-
Convolution	4x4x1	2x2	1	0	ReLU
Max pool	2x2x1	2x2	2	0	-
Flatten	4x1	-	-	-	-
Fully-connected	1	-	-	-	Sigmoid

With the loss function being binary cross entropy loss. The network is set up to do binary classification, i.e., determine if an input image belongs to class 0 or 1. The input image has a single channel and the convolution has depth 1 for making the computation slightly simpler. In total there are 10 trainable parameters ($4 + 1$ in both the convolutional layer and in the fully-connected layer).

(2) An input image X.

Find

1. All itermediate values, output and loss during a forward pass.
2. Gradient of the loss function with respect to weights and biases (both in the fully-connected layer and in the convolutional layer).

Solution

I will use NumPy to do the calculations, and matplotlib to visualize the numbers when performing the forward and backward pass.

In the following code cell I have written all the functions we need to do our computations.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt

def visualize_image(X, label):
    # Plot "pixels" with numerical value and color map
    fig = plt.figure(figsize=(3, 3))
    ax = fig.add_subplot(1,1,1)
    ax.axis('off')
    ax.imshow(X, cmap="coolwarm")
    ax.title.set_text(label)
    for i in range(X.shape[0]):
        for j in range(X.shape[1]):
            ax.text(j, i, f"{X[i,j]:.2f}", color="black", fontsize="small", ha="center", va="center")

def convolution(X, F, b=0, s=1):
    # Convolutional layer
    # note: no padding and depth=1 just to make calculations a bit simpler

    # Output feature size
    w_out = (X.shape[0] - F.shape[0])//s + 1
    h_out = (X.shape[1] - F.shape[1])//s + 1

    z = np.zeros((h_out, w_out))

    for j in range(0, w_out, s):
        for i in range(0, h_out, s):
            z[i,j] = X[i:i+F.shape[1], j:j+F.shape[0]].flatten().dot(F.flatten()) + b
    return z

def max_pool(X, k, s):
    h_out = (X.shape[0] - k)//s + 1
    w_out = (X.shape[1] - k)//s + 1

    z = np.zeros((h_out, w_out))
    idx = []

    for j in range(0, h_out):
        for i in range(0, w_out):
            block = X[j*k:j*k+k, i*k:i*k+k]
            z[j,i] = np.max(block)
```

```
            index = np.add(np.unravel_index(block.argmax(), block.shape), (j*k, i*k))
            idx.append(index) # Save indices of max values for backward pass!
    return z, idx

def ReLU(x):
    return abs(x) * (x > 0) # abs just makes -0.0 into 0.0 for easier reading

def d_ReLU(x):
    return 1 * (x > 0)

def sigmoid(x):
    y = np.exp(x)
    return y / (1 + y)

def d_sigmoid(x):
    y = sigmoid(x)
    return y * (1.0 - y)

def bce_loss(y, p):
    # Binary cross-entropy loss function
    return -(y * np.log10(p) + (1-y) * np.log10(1-p))

def d_bce_loss(y,p):
    # Derivative wrt. p
    return (1 - y) / (1 - p) - (y / p)
```

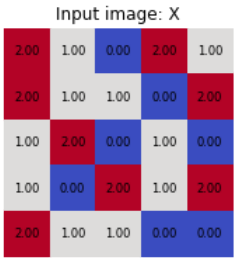
Forward pass

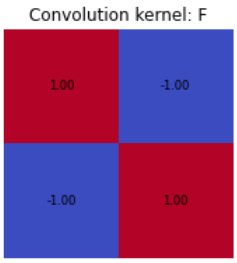
Let X denote the input image assumed to be belonging to class 1:

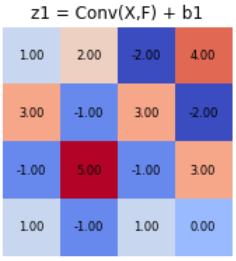
In [2]:
```
X = np.array([[2,1,0,2,1], [2,1,1,0,2], [1,2,0,1,0], [1,0,2,1,2], [2,1,1,0,0]])
y = 1
```

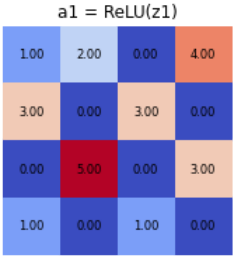
We initialize the weights and bias in the convolutional layer (to some arbitrary values) and perform the forward pass through this layer:

In [3]:
```
# Convolutional layer + ReLU activation function
F = np.array([[1,-1],[-1,1]]) # Kernel
b1 = 1.0 # Bias

z1 = convolution(X, F, b1, s=1) # Perform convolution
a1 = ReLU(z1) # Apply AF

visualize_image(X, "Input image: X")
visualize_image(F, "Convolution kernel: F")
visualize_image(z1, "z1 = Conv(X,F) + b1")
visualize_image(a1, "a1 = ReLU(z1)")
```



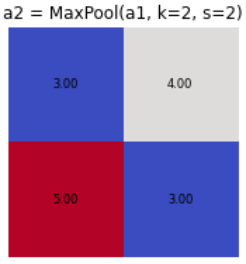


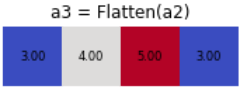




We now apply max pooling and flatten the output of our feature extractor before passing it to the fully-connected layer:

In [4]:
```
# MaxPool + Flatten
a2, max_idx = max_pool(a1, k=2, s=2)
a3 = a2.flatten()

visualize_image(a2, "a2 = MaxPool(a1, k=2, s=2)")
visualize_image(a3.reshape(1,4), "a3 = Flatten(a2)")
```





We initialize the weights and bias of the fully-connected layer and perform the forward pass. We also compute the binary cross-entropy loss:

In [5]:
```
# Fully-connected layer
w4 = np.array([-1, 1, -1, 1]) # Weights
b4 = 0 # Bias

z4 = w4.dot(a3) + b4
a4 = sigmoid(z4)
print(f"Activation after fully-connected layer: {a4:.2f}")

loss = bce_loss(y, a4)
print(f"Binary cross-entropy loss: {loss:.2f}")
```

```
Activation after fully-connected layer: 0.27
Binary cross-entropy loss: 0.57
```

Backward pass

We now perform backpropagation starting with the fully-connected layer. We set

$$\delta := \frac{\partial L}{\partial z_4} = \frac{\partial L}{\partial a_4}\frac{\partial a_4}{\partial z_4} = \frac{\partial L}{\partial a_4}\sigma'(z_4)$$

where σ denotes the sigmoid activation function. Again, by applying the chain rule as needed, we obtain

$$\frac{\partial L}{\partial b_4} = \frac{\partial L}{\partial a_4}\frac{\partial a_4}{\partial z_4}\frac{\partial z_4}{\partial b_4} = \delta \cdot 1 = \delta \quad \text{and similarly} \quad \frac{\partial L}{\partial w_4} = \delta \cdot \frac{\partial z_4}{\partial w_4} = \delta \cdot a_3$$

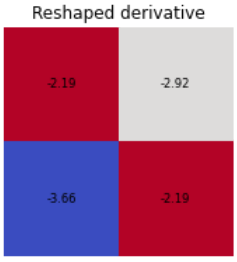
In [6]:
```
lr = 0.1 # Learning rate

# Fully-connected layer
```

```
delta = d_bce_loss(1, a4) * d_sigmoid(z4)
print(f"dL / d(b4) = {delta:.2f}")
#b4 = b4 - lr * delta # Update bias

delta = delta * a3 # Gradient wrt. w4
print(f"dL / d(w4) = {np.round(delta, 2)}")
#w4 = w4 - lr * delta # Update weights
```

```
dL / d(b4) = -0.73
dL / d(w4) = [-2.19 -2.92 -3.66 -2.19]
```

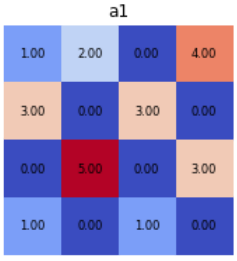
Next, we "unflatten" the backpropagated derivative:

In [7]:
```
delta = delta.reshape(a2.shape)
visualize_image(delta, "Reshaped derivative")
```



We now backpropagate through the max pooling layer. Only the entries appearing as maximum values in the forward pass affects the output of the max pooling layer (with a factor of 1). Consequently, we get that $\frac{\partial L}{\partial a_1}$ can be computed as follows:

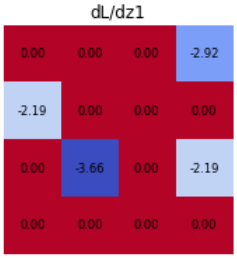
In [8]:
```
d = np.zeros(a1.shape)

for idx, grad in zip(max_idx, delta.flatten()):
    i, j = idx[0], idx[1]
    d[i,j] = grad

delta = d
visualize_image(a1, "a1")
visualize_image(delta, "dL/d(a1)")
```



Next, we backpropagate through the ReLU activation function. The ReLU has slope 1 for non-negative inputs and slope 0 for negative inputs. The derivative is implemented in `d_ReLU()`.

$$\frac{\partial L}{\partial z_1} = \frac{\partial L}{\partial z_1} = \frac{\partial L}{\partial a_1}\frac{\partial a_1}{\partial z_1} = \frac{\partial L}{\partial a_1} \cdot \mathrm{ReLU}'(z_1)$$

In [9]:
```
delta = d_ReLU(z1) * delta
visualize_image(delta, "dL/dz1")
```



Recall that $z_1 = F \odot X + b_1$ (where $\odot$ denotes out convolution operation). In position (i, j) this is given by

$$z_1^{ij} = F_{11}X_{i,j} + F_{12}X_{i,j+1} + F_{21}X_{i+1,j} + F_{22}X_{i+1,j+1} + b_1 \quad \text{for } i, j \in \{1, 2, 3, 4\}.$$

So for F_{kl} we get that

$$\frac{\partial L}{\partial F_{kl}} = \frac{\partial L}{\partial z_1}\frac{\partial z_1}{\partial F_{kl}} = \sum_i \sum_j \frac{\partial L}{\partial z_1^{ij}}\frac{\partial z_1^{ij}}{\partial F_{kl}} \tag{1}$$

$$= \sum_i \sum_j \frac{\partial L}{\partial z_1^{ij}}\frac{\partial}{\partial F_{kl}}(F_{11}X_{i,j} + F_{12}X_{i,j+1} + F_{21}X_{i+1,j} + F_{22}X_{i+1,j+1} + b_1). \tag{2}$$

For each weight in the convolutional layer we then get the following derivatives

$$\frac{\partial L}{\partial F_{11}} = \sum_i \sum_j \frac{\partial L}{\partial z_1^{ij}} X_{i,j}\,, \quad \frac{\partial L}{\partial F_{12}} = \sum_i \sum_j \frac{\partial L}{\partial z_1^{ij}} X_{i,j+1}\,, \quad \frac{\partial L}{\partial F_{21}} = \sum_i \sum_j \frac{\partial L}{\partial z_1^{ij}} X_{i+1,j} \quad \text{and} \quad \frac{\partial L}{\partial F_{22}} = \sum_i \sum_j \frac{\partial L}{\partial z_1^{ij}} X_{i+1,j+1}\,.$$

For the bias b_1 we simply have

$$\frac{\partial L}{\partial b_1} = \sum_i \sum_j \frac{\partial L}{\partial z_1^{ij}}.$$

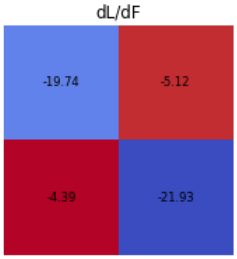
Let us first compute the derivative of L with respect to b_1:

In [10]:
```
grad_b1 = delta.sum() # dL/d(b1)
print(f"dL / d(b1) = {grad_b1:.2f}")
#b1 = b1 - lr * grad_b1 # Update bias
```

```
dL / d(b1) = -10.97
```

We now use the above formulas to compute $\frac{\partial L}{\partial F}$:

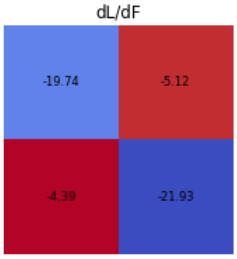
In [11]:
```
grad_F = np.zeros(F.shape)

grad_F[0,0] = np.sum(X[0:4, 0:4] * delta)
grad_F[0,1] = np.sum(X[0:4, 1:5] * delta)
grad_F[1,0] = np.sum(X[1:5, 0:4] * delta)
grad_F[1,1] = np.sum(X[1:5, 1:5] * delta)

# F = F - lr * grad_F # Update weights

visualize_image(grad_F, "dL/dF")
```



Derivation as convolution:

Note that we can also realize the above formulas for computing $\frac{\partial L}{\partial F}$ as a convolution! To be precise, we see that $\frac{\partial L}{\partial F} = X \odot \frac{\partial L}{\partial z_1}$ so we can simplify the code by using convolution:

In [12]:
```
grad_F = convolution(X, delta)

visualize_image(grad_F, "dL/dF")
```


dL/dF
-19.74
-5.12
-4.39
-21.93