Problem description

The following problem is proposed:

6x6 images are generated as one of two distinct classes.

0. Class 0 represents noise as i.i.d. ~ Uniform(low, high), where low and high are parameters that determine the noise level.
1. Class 1 takes class 0 and inserts a 3x3 square by setting each element in a 3x3 grid to a predefined value, starting from a random spot in the image.

The problem then becomes to create a binary classification network that can separate these two.

Given

1. The network architecture

Layer	Input dim	Output dim	Other
Input	6x6	6x6	Class 0 or 1
2D convolve	6x6	4x4	Kernel: 3x3 with stride 1 and no padding. Activation: ReLU
Max pool	4x4	2x2	Kernel: 2x2 with stride 2 and no padding.
Flatten	2x2	4x1	
FCNN	4x1	1	Activation: Sigmoid

2. An input matrix X and a label y, where $y = 0$ when X is all uniform and $y = 1$ when X contains a square.

3. The helper classes Matrix2D and Network

4. An initialized Network object and example data on the Matrix2D format

5. Equations spread throughout with notation as used in the TDT17 lecture slides

Find

1. The intermediate values in both the CNN and FCNN part

2. Post-activation for a forward pass

3. Loss for a forward pass

4. Gradient of the loss function w.r.t. the network parameters

5. The updated network parameters after the backwards pass

Notes

- The helper classes provide a fair bit, making these tasks more of a LEGO set than anything else :)
- Some functions have been borrowed from FCNNs-CNNs-FBPass_Ex1.pdf from the TDT17 Teams page. Other resources are linked in the docstrings of their respective functions. Feel free to use this when solving these tasks.
- All tasks that are to be done are marked with `# TODO:`, so feel free to use the search function.

Imports and helper functions

In [1]:

```python
import numpy as np

import matplotlib.pyplot as plt

from dataclasses import dataclass

from __future__ import annotations
from itertools import product


class Matrix2D():

    def __init__(self, array: np.ndarray):
        assert len(array.shape) == 2, "Input array must be 2D!"

        self.data: np.ndarray = array
        self.nrow: int = array.shape[0]
        self.ncol: int = array.shape[1]

    def __getitem__(self, key):
        return self.data[key]

    def __setitem__(self, key, val):
        self.data[key] = val

    @staticmethod
    def gaussian(center, ndim, sigma = 1):

        xx, yy = np.meshgrid(
            np.linspace(0, ndim-1, ndim) - center[0],
            np.linspace(0, ndim-1, ndim) - center[1]
        )
        return Matrix2D(np.exp(-0.5 * (np.square(xx) + np.square(yy)) / np.square(sigma)))

    @staticmethod
    def uniform_noise(ndim_matrix: int, intensity: float = 1.0) -> Matrix2D:
        return Matrix2D(np.random.uniform(-intensity, intensity, (ndim_matrix, ndim_matrix)))

    @staticmethod
    def square(ndim_matrix: int, ndim_square: int) -> Matrix2D:
        matrix = Matrix2D.uniform_noise(ndim_matrix)

        # Select top left corner to draw square from
        top_left_row = np.random.randint(low=0, high = ndim_matrix - ndim_square)
        top_left_col = np.random.randint(low=0, high = ndim_matrix - ndim_square)

        matrix[top_left_row:top_left_row + ndim_square, top_left_col:top_left_col + ndim_square] = 5.0

        return matrix

    def draw(self, title: str = "") -> None:
        """Display a visualization of the matrix values

        See: https://stackoverflow.com/questions/40887753/display-matrix-values-and-colormap
        """
        fig, ax = plt.subplots()

        ax.matshow(self.data, cmap=plt.cm.Blues)

        if self.nrow == 1:
            for col in range(self.ncol):
                ax.text(col, 0, f"{self.data[0, col]:.2f}", va='center', ha='center')
        elif self.ncol == 1:
            for row in range(self.nrow):
                ax.text(0, row, f"{self.data[row, 0]:.2f}", va='center', ha='center')
        else:
            for col, row in np.ndindex(self.data.shape):
                ax.text(col, row, f"{self.data[row, col]:.2f}", va='center', ha='center')


        ax.set_title(title)

    def convolve2D(self, kernel: Matrix2D, bias: float = 0, stride: int = 1) -> Matrix2D:
        """Convolve the 2D matrix with a 2D kernel plus bias

        See http://www.songho.ca/dsp/convolution/convolution2d_example.html for the math
        """
        assert stride != 0, "Stride cannot be zero!"

        output_width: int = (self.nrow - kernel.nrow) // stride + 1
        output_height: int = (self.ncol - kernel.ncol) // stride + 1

        output_image = np.zeros((output_height, output_width))

        for col, row in product(range(0, output_width, stride), range(0, output_height, stride)):
            output_image[row, col] = \
                self.data[row:row+kernel.ncol, col:col+kernel.nrow].flatten() \
                @ kernel.data.flatten() \
                + bias

        return Matrix2D(output_image)

    def maxpool(self, ndim: int = 1, stride: int = 1):
        """Dimension reduction using the max of a neigborhood defined by ndim

        See https://computersciencewiki.org/index.php/Max-pooling_/_Pooling for the math
        and https://datascience.stackexchange.com/questions/11699/backprop-through-max-pooling-layers
        for the intuition on why the indices of the maximum values for each pool is needed
        """
        assert stride != 0, "Stride cannot be zero!"

        output_width: int = (self.nrow - ndim) // stride + 1
        output_height: int = (self.ncol - ndim) // stride + 1

        output_image = np.zeros((output_height, output_width))
        indices = []

        for row, col in product(range(output_height), range(output_width)):
            rows = slice(row * ndim, (row+1) * ndim)
            cols = slice(col * ndim, (col+1) * ndim)
            pool = self.data[rows, cols]

            output_image[row, col] = np.max(pool)

            index = np.add(np.unravel_index(np.argmax(pool), pool.shape), (row * col, col * row))
            indices.append(index)

        return Matrix2D(output_image), indices

    def flatten(self):
        return Matrix2D(self.data.flatten().reshape(self.nrow*self.ncol, 1))


@dataclass
class Network():
    learning_rate: float = 0.1

    kernel: Matrix2D = None
    conv_bias: float = 0.0

    weights: np.ndarray = np.empty(0)
    fcnn_bias: float = 0.0

    backward: bool = False
    maxpool_indices: np.ndarray = np.empty(0)

    def fcnn_forward(self, x):
        return self.weights @ x + self.fcnn_bias

    def relu(self, value):
        """ReLU activation"""
        act = value > 0

        if self.backward:
            return act * 1

        return act * abs(value)

    def sigmoid(self, value):
        """Sigmoid activation"""
        act = np.exp(value) / (1 + np.exp(value))

        if self.backward:
            return act * (1 - act)

        return act

    def binary_cross_entropy(self, y: float, p: float) -> float:
        """Binary cross entropy loss calculation

        See: https://ml-cheatsheet.readthedocs.io/en/latest/loss_functions.html
        """
        if self.backward:
            return (1 - y) / (1 - p) - (y / p)

        return  - (y * np.log10(p) + (1 - y) * np.log10(1 - p))
```

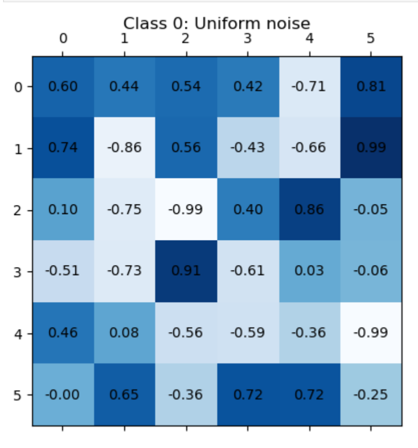
Example of data and usage of the Network class

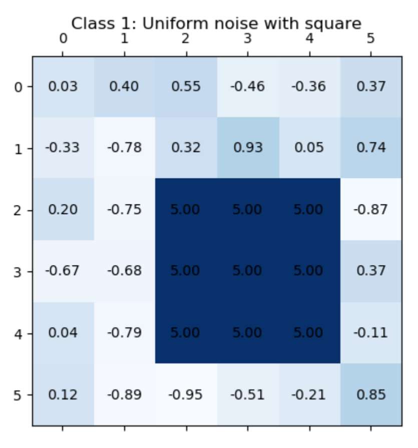
In [2]:

```python
np.random.seed()

# Create Matrix2D for class 0 and 1
ndim = 6

X0 = Matrix2D.uniform_noise(ndim)
X1 = Matrix2D.square(ndim, 3)
X0.draw("Class 0: Uniform noise")
X1.draw("Class 1: Uniform noise with square")
```





Forward pass

The cells below provide an annotated view of the contents of the forward() function of the Network class

We start off by initializing the network parameters. This includes:

1. The 3x3 kernel, initialized with i.i.d. ~ Uniform(-1, 1) elements
2. The convolution bias, initialized as ~ Uniform(0, 1)
3. The FCNN weights, initialized with i.i.d. ~ Uniform(-1, 1) elements
4. The FCNN bias, initialized as ~ Uniform(0, 1)

These are all learnable parameters in the network.

A class 1 example image is also generated for use in the forward and backward pass example.

In [3]:

```python
kernel = Matrix2D(np.random.uniform(low=-1, high=1, size=(3, 3)))
conv_bias = np.random.uniform()

fcnn_weights = np.random.uniform(low=-1, high=1, size=(1, 4))
fcnn_bias = np.random.uniform()

nn = Network(learning_rate=0.01, kernel=kernel, conv_bias=conv_bias, weights=fcnn_weights, fcnn_bias=fcnn_bias)

# Use class 1 as an example
y = 1
X = Matrix2D.square(ndim, 3)
```

Convolution

Since the kernel is initialized to random values, trying to assign any meaning to the output of the convolution layer at this stage does not make sense. It is instead better to understand how the values in the convolved output image are generated. See this gif to get a visualization of the convolution operation.

In [4]:

```python
# TODO
```

ReLU

The ReLU function applies max(0, value) element-wise which adds a nonlinearity to the network. [1] lists some of the benefits of having a ReLU activation, mainly sparsity and fast training times. While neither of these are problematic for such a small-scale problem as here, it yields an extra step to discuss and perform calculations on :)

In [5]:

```python
# TODO
```

Maxpool

Our binary classifier needs a fully-connected neural network between the augmented input matrix and the output. However, due to the sparsity introduced by the ReLU operation, and for scalability concerns, it is reasonable to decimate the augmented image to a lower dimension, and this is what maxpool does. Mathematically, it is similar to convolution in that a sliding window (kernel) is applied to the image. However, instead of weighing the image elements with the kernel data, the maximum image value inside a given window is extracted to the output of the maxpool operation. Again, I refer to an animation that visualizes this.

Instead of requiring 16 weights in our FCNN, we now only need 4 - which is good - from both the perspective of parsimony and computational demand (again, the latter is not *really* a concern for us.)

In [6]:

```python
# TODO
```

FCNN

The FCNN consists of four weights, one for each cell of the maxpool output. In addition, a bias term is introduced to avoid being restricted to only being able to model functions where f(0) = 0

The pre-activation value is calculated by the dot-product between the weights and the input vector, plus bias.

Since the purpose of the network is to discriminate between two classes, a sigmoid activation is applied to the input. This ensures that the output of the network is in the range (0, 1), where values closer to 0 corresponds to the network modelling a given input as class 0, and vice versa.

To assess the performance of the network for a given (known) input, the binary cross-entropy / log-loss function is used, since the output of the sigmoid function may be interpreted as a probability. Log-loss in the binary case is a sum of two parts, each corresponding the the probability of misclassifying that class. Thus, a perfect predictor would have a log-loss of zero.

In [7]:

```python
# TODO
```

Forward pass done - now what?

Through the forward pass, we have accumulated intermediate values, both in the form of scalars (FCNN pre- and post activation, log-loss) and matrices (convolved input, maxpool). The point of doing this is to set up for the *backward pass*, which ultimately leads to a way for us to update the network parameters in such a way as to increase the immediate performance of the network.

Backward pass

The point of the backward pass is to compute the derivative (gradient) of the loss function with respect to the trainable parameters in the network. The intuition for this comes from the fact that, if we are able to tune the network parameters in such a way as to decrease numerical value of the loss function, then we would obtain network parameters that yield "better performance" (as modelled by the loss function).

NOTE: The notation in the math equations match the naming of the intermediate values of the forward pass. Scroll back and forth if you lose track!

In [8]:

```python
# Implementation detail, to ensure that the numerical derivatives of the
# functions in the network are used
nn.backward = True
```

FCNN

For the FCNN layer, we need to calculate the numerical derivative of the loss function w.r.t the weights and the bias of the FCNN, which boils down to smart (or straight-forward) applications of the chain rule.

For the weights:

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial z_4} \frac{\partial z_4}{\partial w}$$

and the bias

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial z_4} \frac{\partial z_4}{\partial b}$$

Noting that

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

where $\frac{\partial L}{\partial a_4}$ is the derivative of the loss function w.r.t. the input, and that $\frac{\partial z_4}{\partial b} = 1$

We get

$$\frac{\partial L}{\partial w} = \delta \cdot a_3, \qquad \frac{\partial L}{\partial b} = \delta$$

NOTE: The "delta" defined in the code from this point on is the delta as defined here, but propagated through the layers of the network, starting at the end.

In [9]:

```python
# TODO
```

Flattening and maxpool

The flattening operation can be reversed without any calculations, as it is a simple reshape of the input vector to the FCNN. The output below shows the reshaped 2x2 derivative.

In [10]:

```python
# TODO
```

Maxpool

Calculating the gradients through the maxpool operation is a bit more involved. It hinges on the fact that, since all non-max values for each window are discarded entirely, the max values are the only ones which we need to calculate the gradient for. That is why the maxpool function borrowed from the published example stores the indices of the max values, so we can insert the gradients from the decimated matrix (above) into a matrix with the same size as the original convolved input. The non-max indices will remain zero, but this is again not a problem since these gradients do not affect the loss.

In [11]:

```python
# TODO
```

ReLU

The gradients propagate through the ReLU activation, which luckily is much simpler than the maxpool, since ReLU is either constant or linear. Mathematically, the gradient at this layer is simply (again using the chain rule)

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

In [12]:

```python
# TODO
```

Convolution

Calculating the derivative of the loss function w.r.t. the convolution kernel does not seem very easy at first, however, by utilizing the fact that convolution can be used to express derivatives, we may use the result

$$\frac{\partial L}{\partial K} = X * \frac{\partial L}{\partial z_1}$$

In [13]:

```python
# TODO
```

Updating network parameters

As mentioned previously, the point of going through all these steps is to update the parameters of the network in such a manner as to decrease the numerical value of the loss function. We have now calculated gradients of the loss function w.r.t. the parameters in the network, and since the gradient is a vector pointing in the steepest (positive) direction at a given point on the manifold spanned by the network parameters, we may negate the gradient to find the *direction of steepest descent*, i.e. do a step of gradient descent. The learning rate, i.e. how far in the direction of steepest descent we perturb the network parameters, is a tunable hyperparameter. Too small, and the learning process takes an infeasibly long time, but too high and the process may overshoot minima and become unstable.

Here it is set arbitrary since "it just worked" :)

In [14]:

```python
# Update weights and biase for convolution
learning_rate = 0.01

# TODO
```

In [ ]: