Lab 7

Emilio Dorigatti

2020-12-18

Welcome to the seventh lab, which is focused on convolutions and convolutional neural networks. The first exercise shows how to train CNNs with Keras, while the second exercise is about implementing convolutions for black-and-white images. The third exercise is about computing the gradients of the convolution operator.

Exercise 1

In this exercise, we will learn how to build CNNs in Keras to classify images.

CNNs are a special type of neural network inspired by the structure of the visual cortex in animals. They can be applied to a wide range of tasks such as image recognition, time-series analysis, sentence classification, etc. Two key features that differentiate CNNs from fully connected nets are:

1. Local connections: Each neuron in a convolutional layer is only connected to a subset of the neurons in the previous layer.
2. Shared weights: Each convolutional layer consists of multiple filters and each filter consists of multiple neurons. All the neurons in a given filter share the same weights but each of these neurons is connected to a different subset of the neurons in the previous layer.

CNNs consistently outperform all other models in machine vision tasks such as image recognition, object detection, etc.

Classifying hand-written digits

We will be working with is the MNIST dataset (included in Keras). It consists of 28x28 pixels, grayscale images of hand-written digits and their associated labels (0 to 9). The training set contains 60,000 images and the test set contains 10,000. Let's load the data:

```r
library(keras)

mnist = dataset_mnist()

#train_images and train_labels form the training set
train_images = mnist$train$x
train_labels = mnist$train$y

#test_images and test_labels from the test set
test_images = mnist$test$x
test_labels = mnist$test$y
```

The images are encoded as 3D arrays of integers from 0 to 255, and the labels are 1D arrays of integers from 0 to 9.

Build the network

A CNN typically consists of a series of convolutional and pooling layers followed by a few fully-connected layers. The convolutional layers detect important visual patterns in the input, and the fully-connected layers then classify the input based on the activations in the final convolutional/pooling layer. Each convolutional layer consists of multiple filters. When the CNN is trained, each filter in a layer specializes in identifying patterns in the image that downstream layers can use.

To create a convolutional layer in Keras, call the `layer_conv_2d` function, and specify the the number of filters in the layer (`filters` parameter), the size of the filters (`kernel_size` parameter), and the activation function to use (`activation` parameter).

Pooling layers are used to downsample intermediate feature maps in the CNN. Keras has multiple options for pooling layers, but today we will only use `layer_max_pooling_2d` which takes a `pool_size` argument for the size of the pooling window.

```r
model = keras_model_sequential() %>%
  layer_conv_2d(
    32, c(3, 3), activation = "relu", input_shape = c(28, 28, 1)
  ) %>%
  layer_max_pooling_2d(
    c(2, 2)
  ) %>%
  layer_conv_2d(
    64, c(3, 3), activation = "relu"
  ) %>%
  layer_max_pooling_2d(
    c(2, 2)
  ) %>%
  layer_conv_2d(
    64, c(3, 3), activation = "relu"
  ) %>%
  layer_flatten() %>%
  layer_dense(
    64, activation = "relu"
  ) %>%
  layer_dense(
    10, activation = "softmax"
  )
```

Let's take a look at what we've built so far:

```r
summary(model)
```

```
## Model: "sequential"
## ________________________________________________________________________________
## Layer (type)                        Output Shape                    Param #
## ================================================================================
## conv2d_2 (Conv2D)                   (None, 26, 26, 32)              320
## ________________________________________________________________________________
## max_pooling2d_1 (MaxPooling2D)      (None, 13, 13, 32)              0
## ________________________________________________________________________________
## conv2d_1 (Conv2D)                   (None, 11, 11, 64)              18496
## ________________________________________________________________________________
## max_pooling2d (MaxPooling2D)        (None, 5, 5, 64)                0
## ________________________________________________________________________________
## conv2d (Conv2D)                     (None, 3, 3, 64)                36928
## ________________________________________________________________________________
## flatten (Flatten)                   (None, 576)                     0
## ________________________________________________________________________________
## dense_1 (Dense)                     (None, 64)                      36928
## ________________________________________________________________________________
## dense (Dense)                       (None, 10)                      650
## ================================================================================
## Total params: 93,322
## Trainable params: 93,322
## Non-trainable params: 0
## ________________________________________________________________________________
```

You can see that the output of every `layer_conv_2d` and `layer_max_pooling_2d` is a 3D tensor of shape `(height, width, channels)`. For example, the output of the first layer is a tensor of shape `(26, 26, 32)`. Note that the width and height dimensions shrink as you go deeper in the network. The number of channels is controlled by the `filters` parameter of the convolutional layers. Also, as you can see, the `(3, 3, 64)` outputs are flattened into vectors of shape $576 = 3 \cdot 3 \cdot 64$ before going through two dense layers.

Data preprocessing

Before training this model, we have to preprocess the data by scaling the inputs. Recall that the inputs are integer arrays in which the elements take values between 0 in 255. It is standard practice to scale the inputs so that the elements take values between 0 and 1. This typically helps the network train better.

```r
# Reshape and rescale train_images and test_images.
train_images = array_reshape(train_images, c(60000, 28, 28, 1))
train_images = train_images / 255

test_images = array_reshape(test_images, c(10000, 28, 28, 1))
test_images = test_images / 255
```

Compile,train and evaluate the model

```r
compile(
  model,
  loss = "sparse_categorical_crossentropy",
  optimizer = "rmsprop",
  metrics = c("accuracy")
)
```

```r
fit(
  model,
  train_images, train_labels,
  batch_size = 64,
  epochs = 5,
  validation_split = 0.2,
  verbose = 0
)
```

```r
evaluate(
  model, test_images, test_labels,
  verbose = 0
)
```

```
## $loss
## [1] 0.03047978
##
## $accuracy
## [1] 0.9905
```

The accuracy of our model on MNIST is quite good!

Exercise 2

In this exercise we are going to implement convolution on images, without worrying about stride and padding, and test it with the Sobel filter. There are two Sobel filters: G_x detects horizontal edges and G_y detects vertical edges.

$$G_x = \begin{vmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{vmatrix} \qquad G_y = \begin{vmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{vmatrix} = {G_x}^T \tag{1}$$

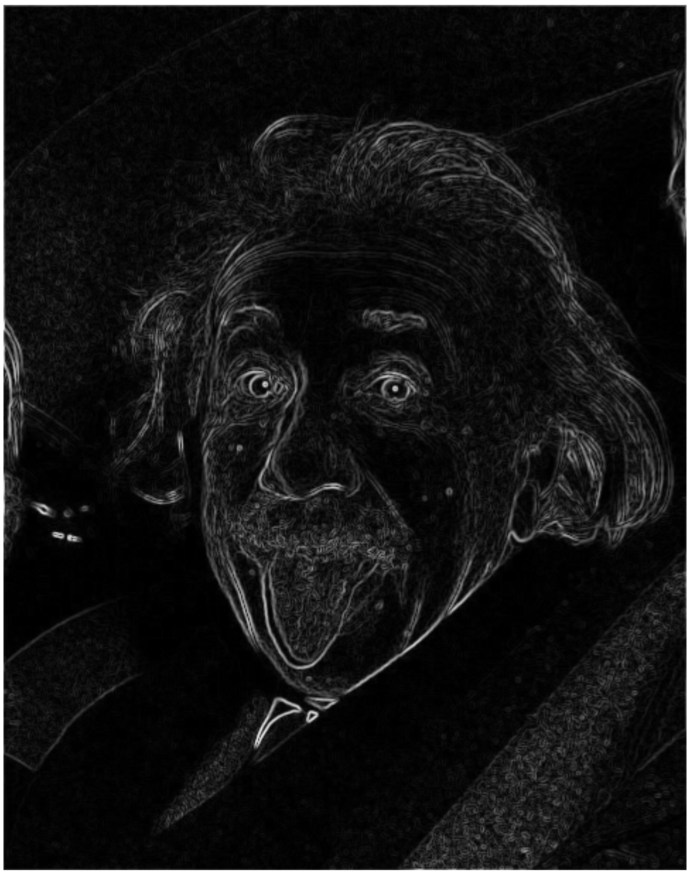
Can you explain why and how these filters work?

In order to get the image E with the edges, we convolve G_x and G_y with the input image I, to obtain the degree of horizontal and vertical "borderness" of each pixel. We then combine these values (separately for each pixel) with an L2 norm:

$$E = \sqrt{(G_x * I)^2 + (G_y * I)^2} \tag{2}$$

As a reference, this is the result we want to obtain:

```r
library(OpenImageR)

# we only keep the first channel, as the image is already black and white
img = readImage("~/Pictures/einstein.jpg")[,,1]

# NB: you can resize the image to speed up subsequent operations
# img = resizeImage(img, floor(ncol(img) / 4), floor(nrow(img) / 4))

# define the two filters
sobel_x = matrix(c(-1, 0, 1, -2, 0, 2, -1, 0, 1), nrow = 3)
sobel_y = matrix(c(-1, -2, -1, 0, 0, 0, 1, 2, 1), nrow = 3)

apply_sobel <- function(conv_fn) {
  # convolve both filters on the image
  conv_x = conv_fn(img, sobel_x)
  conv_y = conv_fn(img, sobel_y)

  # combine the two convolutions
  conv = sqrt(conv_x^2 + conv_y^2)

  # normalize maximum value to 1
  conv / max(conv)
}


result = apply_sobel(convolution)
grid::grid.raster(result)
```



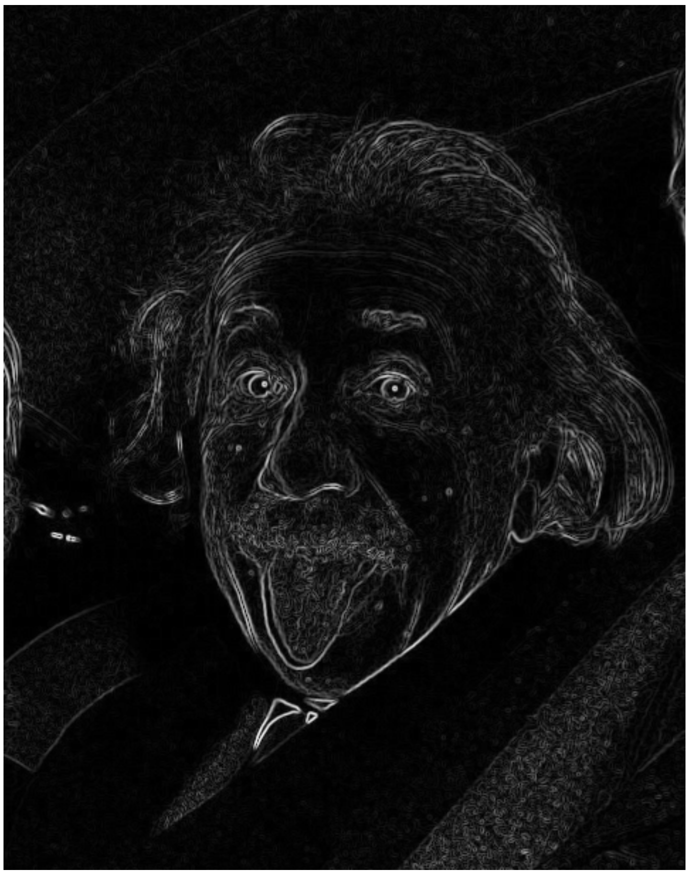
We now implement our version of convolutions. For an input matrix $\mathbf{X}$ of size $r(\mathbf{X}) \times c(\mathbf{X})$ and a kernel $\mathbf{K}$ of size $r(\mathbf{K}) \times c(\mathbf{K})$, the result of the convolution is $\mathbf{Y} = \mathbf{K} * \mathbf{X}$ with $r(\mathbf{Y}) = r(\mathbf{X}) - r(\mathbf{K}) + 1$, $c(\mathbf{Y}) = c(\mathbf{X}) - c(\mathbf{K}) + 1$, and elements:

$$y_{ij} = \sum_{k=1}^{r(\mathbf{K})} \sum_{l=1}^{c(\mathbf{K})} x_{i+k-1,j+l-1} \cdot k_{kl} \tag{3}$$

for $1 \leq i \leq r(\mathbf{Y})$ and $1 \leq j \leq c(\mathbf{Y})$.

You now have to implement a function that computes y_{ij} given the image, the kernel, i and j.

```
compute_convolution_at_position <- function(i, j, image, kernel) {
  result_ij = 0.0
  for(k in 1:nrow(kernel)) {
    for(l in 1:ncol(kernel)) {
      result_ij = result_ij + image[i + k - 1, j + l - 1] * kernel[k, l]
    }
  }
  result_ij
}


our_conv <- function(image, kernel) {
  result_rows = (
    nrow(image) - nrow(kernel) + 1
  )

  result_cols = (
    ncol(image) - ncol(kernel) + 1
  )
  # perform the convolution
  vec = apply(expand.grid(1:result_rows, 1:result_cols), 1, function(pos) {
    compute_convolution_at_position(pos[1], pos[2], image, kernel)
  })

  # reshape to a matrix
  matrix(vec, nrow = result_rows, ncol = result_cols)
}

our_result = apply_sobel(our_conv)
grid::grid.raster(our_result)
```



If you did everything correctly, this image should match the image above.

Exercise 3

Recall that the convolution $\mathbf{Y} = \mathbf{K} * \mathbf{X}$ has elements

$$y_{ij} = \sum_{k=1}^{r(\mathbf{K})} \sum_{l=1}^{c(\mathbf{K})} x_{i+k-1,j+l-1} \cdot k_{kl} \tag{4}$$

Now consider $\mathbf{X}$ and $\mathbf{Y}$ to be the input and output of a convolutional layer with filter $\mathbf{K}$. For simplicity, we focus on a single channel; actual convolution layers in CNN perform this operation several times with different learnable filters.

Imagine this convolution is a hidden layer of the neural network, with $\mathbf{X}$ being the input from the previous layer, and $\mathbf{Y}$ the pre-activation output to the next layer. Then, we can define the loss function in terms of $\mathbf{Y}$, i.e. $\mathcal{L} = f(\mathbf{Y})$, where f includes the activation, all the following layers, and the classification/regression loss.

Show that:

$$\frac{\partial \mathcal{L}}{\partial k_{kl}} = \sum_{i=1}^{r(\mathbf{Y})} \sum_{j=1}^{c(\mathbf{Y})} \frac{\partial \mathcal{L}}{\partial y_{ij}} \cdot x_{i+k-1,j+l-1} \tag{5}$$

Then show that

$$\frac{\partial \mathcal{L}}{\partial x_{ij}} = \sum_{k=L_k}^{U_k} \sum_{l=L_l}^{U_l} \frac{\partial \mathcal{L}}{\partial y_{ab}} k_{kl} \tag{6}$$

With

$$a = i - k + 1 \tag{7}$$
$$b = j - l + 1 \tag{8}$$
$$L_k = \max(1, i - r(\mathbf{X}) + r(\mathbf{K})) \tag{9}$$
$$L_l = \max(1, j - c(\mathbf{X}) + c(\mathbf{K})) \tag{10}$$
$$U_k = \min(r(\mathbf{K}), i) \tag{11}$$
$$U_l = \min(c(\mathbf{K}), j) \tag{12}$$

As you can see, the gradient of the input is obtained by convolving the same filter with the gradient of the output, with some care at the borders.

Hint: it is easier to analyze convolutions in one dimension with a small example, then generalize the result to two dimensions and arbitrary filter/image size.

Now, write a function that computes $\partial \mathcal{L}/\partial x_{ij}$, with $\mathcal{L} = \sum_{i,j} {y_{ij}}^2$ and $\mathbf{K} = G_x$.

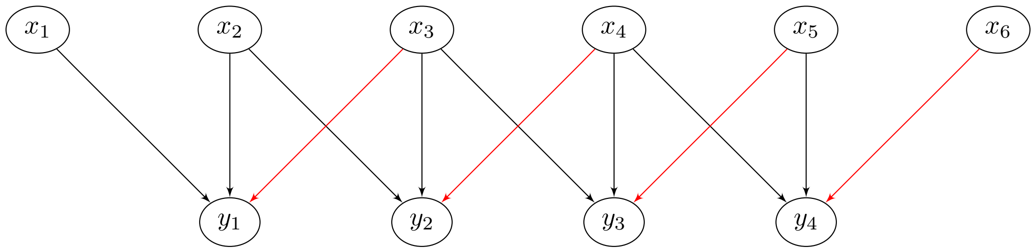
Solution We start by showing that

$$\frac{\partial \mathcal{L}}{\partial k_{kl}} = \sum_{i=1}^{r(\mathbf{Y})} \sum_{j=1}^{c(\mathbf{Y})} \frac{\partial \mathcal{L}}{\partial y_{ij}} \cdot x_{i+k-1,j+l-1} \tag{13}$$

The loss is a generic function of every y_{ij}, therefore by the chain rule we have

$$\frac{\partial \mathcal{L}}{\partial k_{kl}} = \sum_{i=1}^{r(\mathbf{Y})} \sum_{j=1}^{c(\mathbf{Y})} \frac{\partial \mathcal{L}}{\partial y_{ij}} \cdot \frac{\partial y_{ij}}{\partial k_{kl}} \tag{14}$$

The element of the kernel is fixed (because k and l are given), and each y_{ij} is influenced only once by k_{kl}. This means that we have to find which element of $\mathbf{X}$ is $\partial y_{ij}/\partial k_{kl}$. In the image below, $\mathbf{X}$ is a vector of $c(\mathbf{X}) = 6$ elements, and $\mathbf{Y}$ was obtained with a kernel of size $c(\mathbf{K}) = 3$, which result in $c(\mathbf{Y}) = 4$. The arrows indicate which elements of $\mathbf{X}$ were used to compute which element of $\mathbf{Y}$, and the red arrows highlight the third element of the kernel, i.e. k_3.



We can see that $\partial y_i/\partial k_3 = x_{i+3-1}$, e.g. $\partial y_2/\partial k_3 = x_4$, so that:

$$\frac{\partial \mathcal{L}}{\partial k_i} = \sum_{j=1}^{6} \frac{\partial \mathcal{L}}{\partial y_j} \cdot x_{i+j-1} \tag{15}$$

This extends straightforwardly to several dimensions.

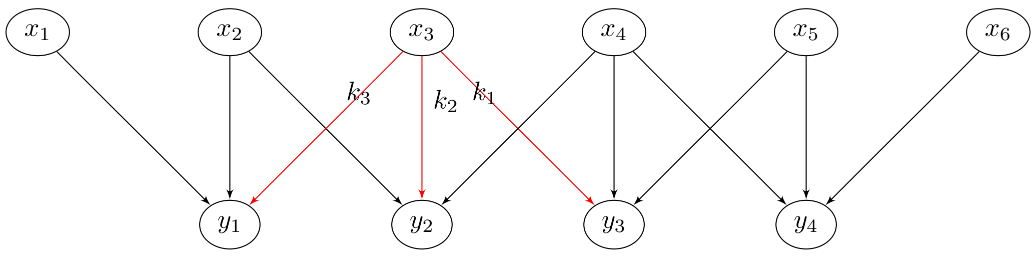
We now show that

$$\frac{\partial \mathcal{L}}{\partial x_{ij}} = \sum_{k=L_k}^{U_k} \sum_{l=L_l}^{U_l} \frac{\partial \mathcal{L}}{\partial y_{ab}} k_{kl} \tag{16}$$

With

$$a = i - k + 1 \tag{17}$$
$$b = j - l + 1 \tag{18}$$
$$L_k = \max(1, i - r(\mathbf{X}) + r(\mathbf{K})) \tag{19}$$
$$L_l = \max(1, j - c(\mathbf{X}) + c(\mathbf{K})) \tag{20}$$
$$U_k = \min(r(\mathbf{K}), i) \tag{21}$$
$$U_l = \min(c(\mathbf{K}), j) \tag{22}$$

Going back to the example in one dimension, we now highlight the influence of a fixed element of the input, such as x_3:



We also annotated which element of the kernel is used in every edge.

For brevity, let $y'_i = \partial \mathcal{L} / \partial y_i$. Then, from the image we see that:

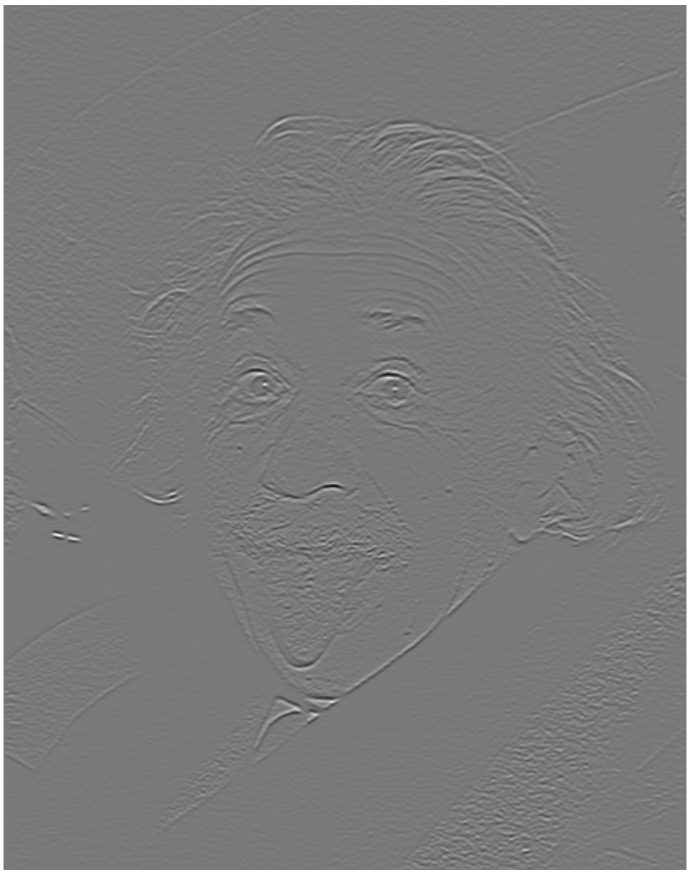
$$\frac{\partial \mathcal{L}}{\partial x_3} = y'_3 k_1 + y'_2 k_2 + y'_1 k_3 = \sum_{i=1}^{3} y'_{3-i+1} k_i \tag{23}$$

In particular, note that the indices increse on $\mathbf{K}$ and decrease on $\mathbf{Y}$. Some care needs to be taken at the borders:

$$\frac{\partial \mathcal{L}}{\partial x_1} = k_1 y'_1$$
$$\frac{\partial \mathcal{L}}{\partial x_2} = k_2 y'_1 + k_1 y'_3$$
$$\frac{\partial \mathcal{L}}{\partial x_5} = k_2 y'_4 + k_3 y'_3$$
$$\frac{\partial \mathcal{L}}{\partial x_6} = k_3 y'_4$$

In general, the first and last $c(\mathbf{K}) - 1$ elements will have less summands, i.e. when $i < c(\mathbf{K})$ and $i > c(\mathbf{X}) - c(\mathbf{K}) + 1$. In the former case, we sum the kernel elements from 1 to i, and in the latter case we sum from $i - c(\mathbf{X}) + c(\mathbf{K})$ to 3 (both ends included). As before, this reasoning extends trivially to multiple dimensions.

```r
conv_gradient_wrt_input <- function(dloss_dy, kernel) {
  image_rows = (
    nrow(dloss_dy) + nrow(kernel) - 1
  )

  image_cols = (
    ncol(dloss_dy) + ncol(kernel) - 1
  )

  vec = apply(expand.grid(1:image_rows, 1:image_cols), 1, function(pos) {
    i = pos[1]
    j = pos[2]

    lk = max(1, i - image_rows + nrow(kernel))
    ll = max(1, j - image_cols + ncol(kernel))
    uk = min(nrow(kernel), i)
    ul = min(ncol(kernel), j)

    gradient_ij = 0.0
    for(k in lk:uk) {
      for(l in ll:ul) {
        a = i - k + 1
        b = j - l + 1

        gradient_ij = gradient_ij + dloss_dy[a, b] * kernel[k, l]
      }
    }
    gradient_ij
  })

  matrix(vec, nrow = image_rows, ncol = image_cols)
}


result = our_conv(img, sobel_x)
input_gradient = conv_gradient_wrt_input(2 * result, sobel_x)
grid::grid.raster((
  input_gradient - min(input_gradient)
) / (
  max(input_gradient) - min(input_gradient)
))
```



We can verify this gradient is correct for a single pixel with finite differences:

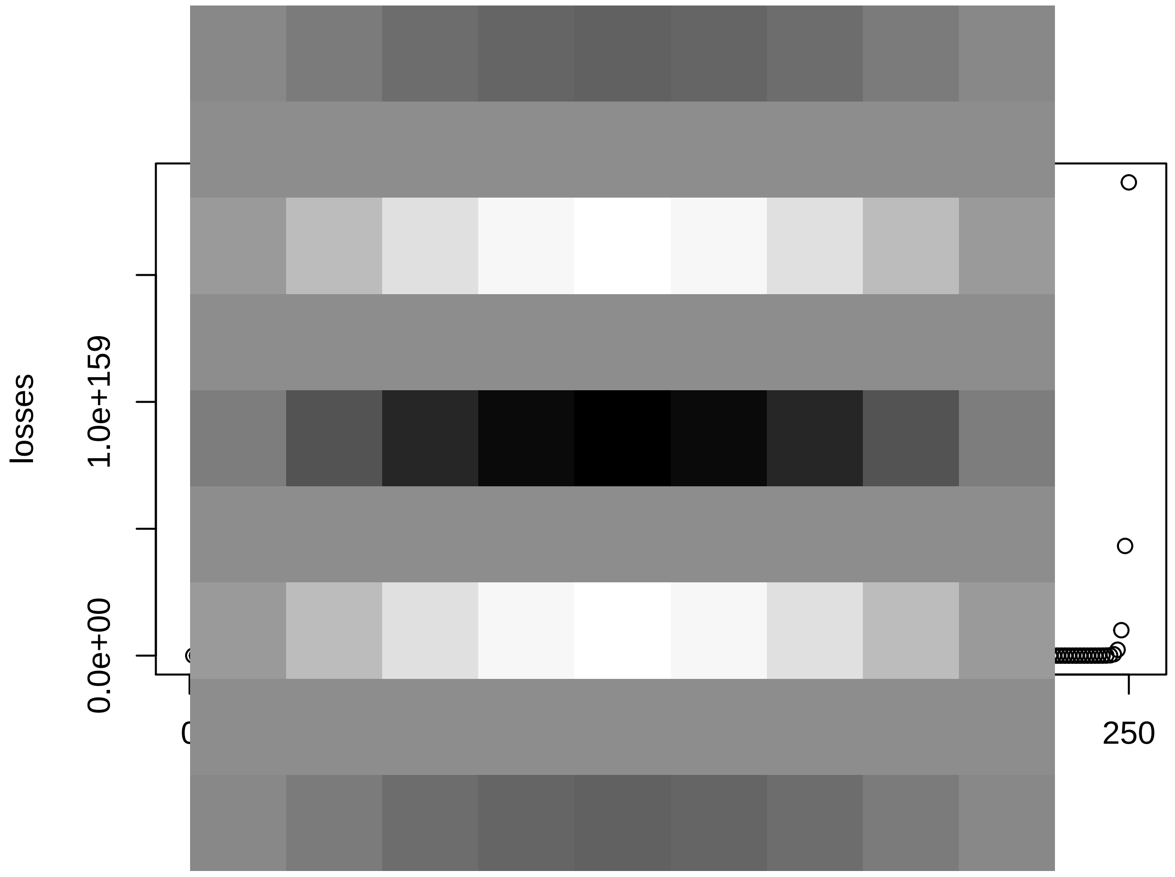
```
eps = 1e-6

i = floor(runif(1, 1, nrow(img) + 1))
j = floor(runif(1, 1, ncol(img) + 1))

# add epsilon to position i,j and convolve
img[i, j] = img[i, j] + eps
conv_pos = our_conv(img, sobel_x)

# remove epsilon to position i,j and convolve
img[i, j] = img[i, j] - 2 * eps
conv_neg = our_conv(img, sobel_x)

# undo modification to the image
img[i, j] = img[i, j] + eps

# compute the difference of the losses
# NB: we sum the differences to get a more accurate result
empirical_gradient = sum(conv_pos^2 - conv_neg^2) / (2 * eps)

# compare empirical and analytical gradients
c(empirical_gradient, input_gradient[i, j])
```

```
## [1] -0.972549 -0.972549
```

If you did everything correctly, these two numbers should be the same.

Now, can you guess what image *maximizes* the loss we just defined? We can find this through gradient *ascent*:

```
maxim = matrix(runif(81), nrow = 9)

losses = lapply(1:250, function(i) {
  conv = our_conv(maxim, sobel_x)
  loss = sum(conv^2)

  grad = conv_gradient_wrt_input(2 * conv, sobel_x)
  maxim <<- maxim + 0.01 * grad

  loss
})

plot(1:length(losses), losses)
grid::grid.raster(
  (maxim - min(maxim)) / (max(maxim) - min(maxim)),
  interpolate = FALSE)
```


losses
1.0e+159
0.0e+00