Lab 12

Emilio Dorigatti

2021-02-05

Exercise 1

```
library(tensorflow)
cat("This code was tested on tensorflow version", tf$version$GIT_VERSION)
```

```
## This code was tested on tensorflow version v2.1.0-rc2-17-ge5bf8de
```

In this lab we are going to train a generative adversarial network (GAN) on the MNIST dataset using Keras. A useful set of tricks to train GANs can be found at https://github.com/soumith/ganhacks.

As usual, we start by loading and normalizing the MNIST dataset but, this time, we normalize values to $[-1, 1]$.

```
library(keras)

mnist = dataset_mnist()
x_train = (mnist$train$x - 127.5) / 127.5
y_train = to_categorical(mnist$train$y)
input_size = 28 * 28
dim(x_train) <- c(nrow(x_train), input_size)
```

Next, we create the generator network. This time, we are going to use an optimizer with customized parameters. Specifically, we use the Adam optimizer with a learning rate of 0.0002 and $\beta_1 = 0.5$. This will give more weights to recent gradient updates.

The generator takes as input normally-distributed, 32-dimensional noise vectors and should be composed of four fully-connected layer of sizes 256, 512, 1024 and 784, each using a leaky ReLU activation with $\alpha = 0.2$ except for the output layer which uses tanh. Use a mean squared error loss for the generator.

```
optim = optimizer_adam(
  lr=0.0002, beta_1=0.5
)

noise_size = 32
generator = keras_model_sequential() %>%
  layer_dense(256, input_shape = c(noise_size)) %>%
  layer_activation_leaky_relu(alpha = 0.2) %>%
  layer_dense(512) %>%
  layer_activation_leaky_relu(alpha = 0.2) %>%
  layer_dense(1024) %>%
  layer_activation_leaky_relu(alpha = 0.2) %>%
  layer_dense(784, activation = "tanh")

generator %>% compile(loss = "mse", optimizer = optim)
```

We now create the discriminator with a mirrored architecture, i.e. layers of size 1024, 512, 256 and 1. As before, use the leaky ReLU activation with $\alpha = 0.2$, and add dropout with $p = 0.3$ between all layers. As the discriminator will perform a binary classification task, use the sigmoid activation in the output layer and the binary cross-entropy loss. Use the same optimizer you created earlier.

```r
discriminator = keras_model_sequential() %>%
  layer_dense(1024, input_shape = c(784)) %>%
  layer_activation_leaky_relu(alpha = 0.2) %>%
  layer_dropout(0.3) %>%
  layer_dense(512) %>%
  layer_activation_leaky_relu(alpha = 0.2) %>%
  layer_dropout(0.3) %>%
  layer_dense(256) %>%
  layer_activation_leaky_relu(alpha = 0.2) %>%
  layer_dropout(0.3) %>%
  layer_dense(1, activation = "sigmoid")

discriminator %>% compile(loss = "binary_crossentropy", optimizer = optim)
```

We can now use the generator and the discriminator separately. However, during training, we need to pass through the discriminator to compute the gradients of the loss with respect to the generator. For this reason, we need to create a single model comprising both generator and discriminator:

```r
gan_input = layer_input(shape = c(noise_size))
gan_output = discriminator(generator(gan_input))
gan = keras_model(gan_input, gan_output)
gan %>% compile(loss = "binary_crossentropy", optimizer = optim)
```

We are now ready to train our generative adversarial network. In each training step, we first train the discriminator alone for five batches, using both real and generated images, then we perform a single update on the discriminator and generator together. Freezing the discriminator weights is an option that can make training easier, but we will not do this here.

Because of this particular training procedure, we need to write our own optimization loop. Keras models provide a method, `train_on_batch` (documentation here), to execute a single weight update based on an individual batch.

Note that the loss of the `gan` model will be used whenever we use its `fit` or `train_on_batch` methods; the losses for the generator and discriminator will not be used. Since we never use the generator directly, its mean squared error loss is never really applied. It is there only because we have to specify a loss in order to compile a model.

```r
batch_size = 128

generator_losses = c()
discriminator_losses = c()

for(i in 1:2500) {
  for (k in 1:5) {
    batch_idx = sample(1:nrow(x_train), batch_size)
    batch_images = x_train[batch_idx,]

    noise = matrix(rnorm(batch_size * noise_size), nrow = batch_size)

    generated = generator$predict(noise)

    dloss = (
      discriminator$train_on_batch(
        rbind(batch_images, generated),
        c(rep(0.9, batch_size), rep(0.1, batch_size))
      )
    )
  }

  gloss = (
    gan$train_on_batch(
      matrix(rnorm(batch_size * noise_size), nrow = batch_size),
      rep(1, batch_size)
    )
  )

  # store losses
  discriminator_losses = c(discriminator_losses, dloss)
  generator_losses = c(generator_losses, gloss)

  # print progress message
  if(i %% 250 == 0) {
    l = length(generator_losses)
    cat(
      "===  step", i,
      "\n     generator loss",
      mean(generator_losses[l-99:l]),
      "\n     discriminator loss",
      mean(discriminator_losses[l-99:l]),
      "\n"
    )
  }
}
```

```
## ===  step 250
##      generator loss 0.7628276
##      discriminator loss 0.5513438
## ===  step 500
##      generator loss 0.7938182
##      discriminator loss 0.5624696
## ===  step 750
##      generator loss 0.8048381
##      discriminator loss 0.561826
## ===  step 1000
##      generator loss 0.807338
##      discriminator loss 0.5618195
## ===  step 1250
##      generator loss 0.8066306
##      discriminator loss 0.5630467
## ===  step 1500
##      generator loss 0.8042434
##      discriminator loss 0.5649999
## ===  step 1750
##      generator loss 0.8014263
##      discriminator loss 0.5670402
## ===  step 2000
##      generator loss 0.7977408
##      discriminator loss 0.5692115
## ===  step 2250
##      generator loss 0.7943721
##      discriminator loss 0.5711928
## ===  step 2500
##      generator loss 0.7911328
##      discriminator loss 0.5731165
```

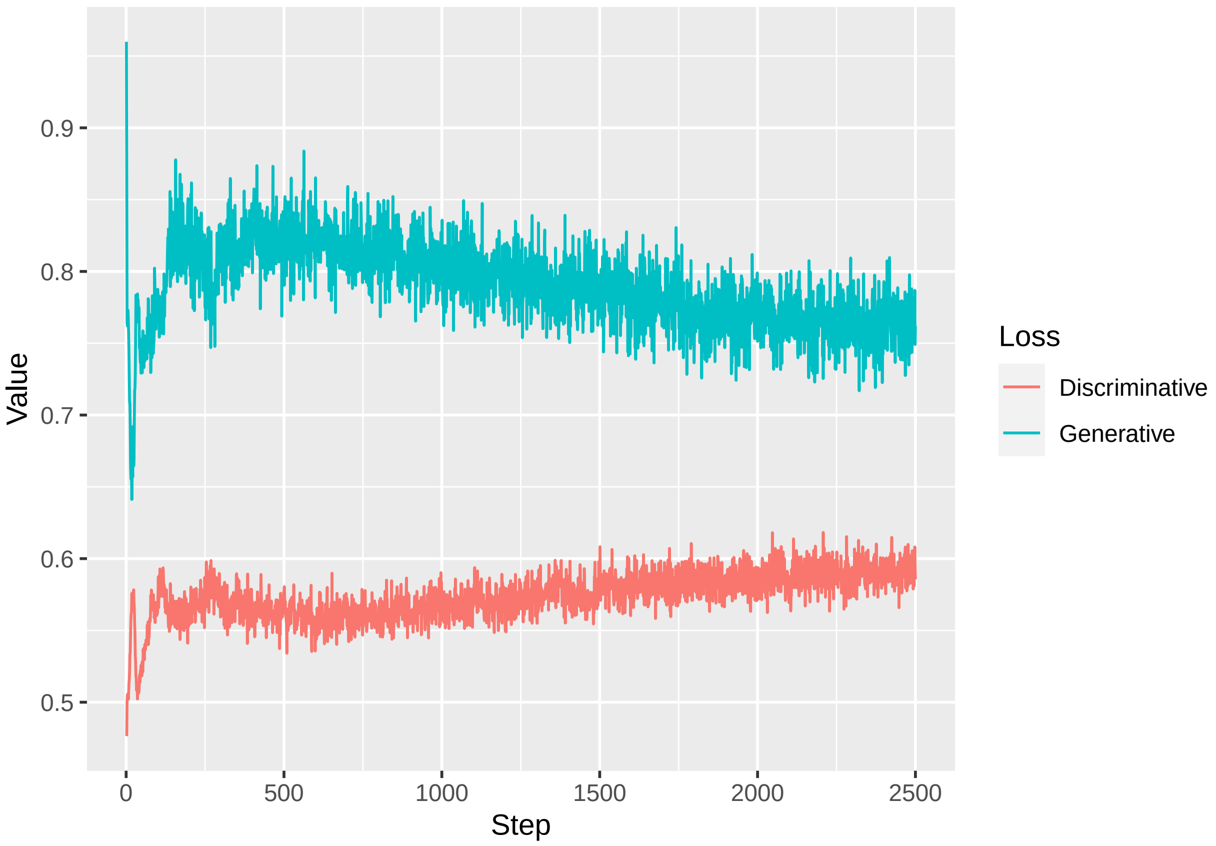
```
# ballpark ranges for acceptable losses
stopifnot(dloss > 0.5 && dloss < 0.7 && gloss > 0.7 && gloss < 1.0)
```

This is how the losses evolved during training:

```
library(ggplot2)

df = rbind(
  data.frame(
    Step = 1:length(generator_losses),
    Value = generator_losses,
    Loss = "Generative"
  ),
  data.frame(
    Step = 1:length(discriminator_losses),
    Value = discriminator_losses,
    Loss = "Discriminative"
  )
)

ggplot(df, aes(x = Step, y = Value, colour = Loss)) +
  geom_line()
```



Training did not fully converge, as evidenced by the generative loss still decreasing and the discriminative loss increasing at the end of training.

Finally, we can try to use the trained GAN to generate new samples by feeding random noise to the generator network:

```
noise = matrix(rnorm(batch_size * noise_size), nrow = batch_size)
generated = generator$predict(noise)
```

This is a sample of nine generated images:

```
library(grid)

lay <- grid.layout(3, 3)
vplay <- viewport(layout=lay)
```

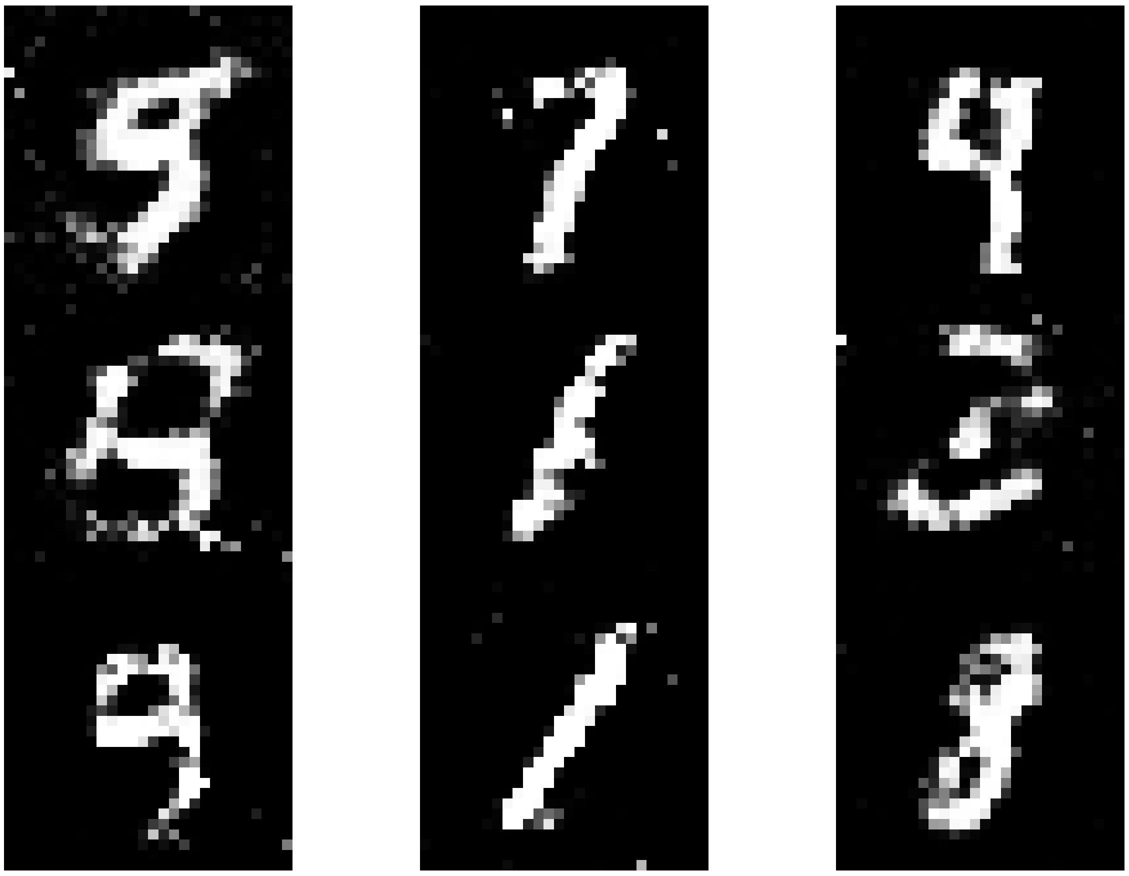
```r
pushViewport(vplay)

for(i in 0:8) {
  # transform vector to matrix
  gen = matrix(generated[i + 1,] / 2 + 0.5, nrow = 28)

  # pad values
  gen[gen < 0] = 0
  gen[gen > 1] = 1

  # select appropriate location in the chart grid
  pushViewport(viewport(
    layout.pos.row=as.integer(i / 3) + 1,
    layout.pos.col=i %% 3 + 1)
  )
  # show the image
  grid.raster(gen, interpolate = FALSE)

  upViewport()
}
```



Although not perfect, these images certainly resemble MNIST digits! Had we trained for longer and/or used a better model, we would certainly be able to generate perfect-looking MNIST digits.