Lab 11

Emilio Dorigatti

2021-01-29

Exercise 1

In this exercise we will get acquainted with the KL divergence for normal distributions. First, let $p(x) = \mathcal{N}(\mu_1, \sigma_1^2)$ and $q(x) = \mathcal{N}(\mu_2, \sigma_2^2)$ and show that

$$\mathrm{KL}(q||p) = \mathbb{E}_{x\sim q}\left[\log \frac{q(x)}{p(x)}\right] = \log \frac{\sigma_1}{\sigma_2} + \frac{\sigma_2^2 + (\mu_1 - \mu_2)^2}{2\sigma_1^2} - \frac{1}{2} \tag{1}$$

Now, consider a variaitonal autoencoder that takes a vector as input $\mathbf{x}$ and transforms it into a mean vector $\mu(\mathbf{x})$ and a variance vector $\sigma(\mathbf{x})^2$. From these, we derive the latent code $\mathbf{z} \sim q(\mathbf{z}) = \mathcal{N}(\mu(\mathbf{x}), \mathrm{diag}(\sigma(\mathbf{x})^2))$, i.e. a multivariate Gaussian in d dimensions with a given mean vector and diagonal covariance matrix. The prior distribution for $\mathbf{z}$ is another d-dimensional multivariate Gaussian $p = \mathcal{N}(\mathbf{0}, \mathbf{1})$.

Now show that:

$$\mathrm{KL}(q||p) = -\frac{1}{2}\sum_{i=1}^{d}\left(1 + \log \sigma_i(\mathbf{x})^2 - \sigma_i(\mathbf{x})^2 - \mu_i(\mathbf{x})^2\right) \tag{2}$$

Hint: start by showing that p and q can be factorized into a product of independent Gaussian components, one for each dimension, then apply the formula for the KL divergence for the univariate case.

Exercise 2

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

In this exercise we are going to implement variational autoencoders (VAEs) on the MNIST dataset. In doing this, we will learn some advanced features of Keras that allow us to extend its capabilities beyond what is available out of the box. Specifically, we will see how to create custom layers that can be integrated seamlessly with the other components, and how to create custom callbacks that monitor and alter the training process.

As usual, we start by loading and pre-processing the dataset.

```
library(keras)

mnist = dataset_mnist()
x_train = mnist$train$x / 255
y_train = to_categorical(mnist$train$y)
input_size = 28 * 28
dim(x_train) <- c(nrow(x_train), input_size)
```

In a VAE, the encoder outpus mean and variance of a multivariate Gaussian distribution of the latent codes. Nothing prevents you from using a more complicated distribution in the same framework, but this is the usual choice. The expected log likelihood is then approximated by decoding a single sample from this distribution. Moreover, since we need the model to be differentiable end-to-end, sampling from the latent codes is re-formulated via the reparametrization trick.

Keras does not provide a layer that can perform this sampling operation, but it can be easily extended to include such a layer. This involves creating a R6 that implements these three methods:

1. `build(input_shape)`: in this method, you create the parameters necessary for the layer to operate.
2. `call(x)`: this method contains the computational logic of the layer, and computes the output from the input `x` of the previous layer.
3. `compute_output_shape(input_shape)`: this computes the size of the output.

In practice, these functions operate with Tensorflow variables, although it is not immediately clear; `build` creates them, and `call` creates the part of the computational graph responsible to compute the output.

Finally, you have to provide a function that is used to instantiate the layer, similar to `layer_dense`, `layer_conv_2d`, etc.

For example, here is how to implement a normal dense layer:

```
OurDense <- R6::R6Class(
  "OurDense",

  inherit = KerasLayer,

  public = list(
    # these variables contain the attributes of this layer
    units = NULL,
    activation = NULL,
    weights = NULL,
    bias = NULL,

    # this function receives the parameters for the layer
    # and saves them in the variables above. in order to access
    # these variables, you must always use "self$" before their name
    initialize = function(units, activation) {
      self$units = units
      self$activation = activation
    },

    # this function creates the weight matrix and the bias vector.
    # keras conveniently provides an utility function to do so, `add_weight`,
    # which can accept other parameters, such as regularizers, etc.
    build = function(input_shape) {
      self$weights <- self$add_weight(
        name = 'weights',
        shape = list(input_shape[[2]], self$units),
        initializer = initializer_glorot_normal(),
      )

      self$bias <- self$add_weight(
        name = 'bias',
        shape = list(self$units),
        initializer = initializer_constant(),
      )
    },

    # this function computes the output from the input.
    # Keras provides functions to do this through the so-called
    # "backend", which is just a wrapper around Tensorflow.
    # functions in this backend all start with "k_".
    # you can also directly use the functions in Tensorflow.
    call = function(x, mask = NULL) {
      self$activation(k_dot(x, self$weights) + self$bias)
```

```
    },

    # this function computes the output shape
    compute_output_shape = function(input_shape) {
      list(input_shape[[1]], self$units)
    }
  )
)

# define layer wrapper function. the parameter `object` refers to the previous layer,
# while the other parameters are defined by the user.
# here you have to gather all parameters that are needed for the `initialize` method in
# the class above, as well as the input shape.
layer_our_dense <- function(object, units, activation, input_shape = NULL) {

  # `create_layer` function is provided by keras, we pass the layer class, the previous
  # object in the network, and a list of parameters to create the layer.
  create_layer(OurDense, object, list(
    units = as.integer(units),  # do not forget
    activation = activation,
    input_shape = input_shape
  ))
}
```

Now we can use this as any other Keras layer:

```
model <- keras_model_sequential() %>%
  layer_our_dense(units = 32, activation = activation_relu,
                  input_shape = tf$TensorShape(as.integer(784))) %>%
  layer_our_dense(units = 10, activation = activation_softmax)

model %>% compile(
  loss = 'categorical_crossentropy',
  optimizer = 'adam',
  metrics = c('accuracy')
)

model %>% fit(
  x_train, y_train,
  epochs = 2
)
```

We now create a custom layer that performs sampling for a VAE. Until now, we have used `keras_model_sequential` to build our neural networks, but Keras can also build networks that are a general directed acyclic graph. In this graph, every node is a whole layer, and a layer can process data from several other layers, and can send its output to an arbitrary number of other layers. We will use this to separate the creation of mean and standard deviation for the latent code, and write our custom layer to take both of them as input. Moreover, for numerical reasons, the encoder network will actually predict the logarithm of the standard deviation.

Let us start from the custom layer.

```
# this parameter re-weighs the KL divergence
# later it will be clear why it is needed
# this is actually a tensorflow variable
kl_weight = k_variable(1.0)

SamplingLayer <- R6::R6Class("SamplingLayer",

  inherit = KerasLayer,
```

```
  public = list(
    # we do not need any parameters for this layer
    initialize = function() { },

    build = function(input_shape) { },

    call = function(x, mask = NULL) {
      # here `x` is a list of two matrices,
      # the mean and log variance of the latent codes.
      # (samples are on rows and latent dimensions on columns)
      mu = x[[1]]
      log_var = x[[2]]

      kl_loss = (
        # TODO compute the KL divergence according to equation 2
        # some useful functions are k_sum, k_exp and k_square
      )

      self$add_loss(
        kl_weight * kl_loss / 784  # scale the KL to a magnitude comparable to the MSE
      )

      # generate the random noise
      eps = k_random_normal(k_shape(mu))

      # TODO compute the latent code using the reparametrization trick
    },

    compute_output_shape = function(input_shape) {
      # `input_shape` is a list of lists, i.e.
      # list(list(batch_size, mean_size), list(batch_size, std_size))
      list(input_shape[[1]])
    }
  )
)

layer_sampling <- function(object) {
  create_layer(SamplingLayer, object, list())
}
```

This encoder has two hidden layers of size 256 and 64. The output of the latter is connected to two separate layers that will predict mean and (the logarithm of the) standard deviation of the latent code for the input samples. We now connect these two layers to the sampling layer we implemented above, and add a few more dense layers to reconstruct the example.

We now create the model, starting from the encoder.

```
latent_dim = 32
hidden_dim = 512
sample_dim = 784

input  = layer_input(shape = list(sample_dim))

# common encoder layers
ll = input %>% layer_our_dense(hidden_dim, activation_relu)

# separate heads for mean and standard deviation
latent_mean = ll %>% layer_our_dense(latent_dim, activation_linear)
```

```
latent_log_std = ll %>% layer_our_dense(latent_dim, activation_linear)
```

This encoder has two hidden layers of size 256 and 64. The output of the latter is connected to two separate layers that will predict mean and (the logarithm of the) standard deviation of the latent code for the input samples. We now connect these two layers to the sampling layer we implemented above, and add a few more dense layers to reconstruct the example.

```
# note that the input is a list of two layers
latent_code = layer_sampling(list(latent_mean, latent_log_std))

# decoder layers
output = latent_code %>%
  layer_our_dense(hidden_dim, activation_relu) %>%
  layer_our_dense(sample_dim, activation_linear)
```

We can create a Keras model with arbitrarily complicated topologies by appropriately connecting layers, as we just did. We only need to tell Keras which layers are input layers and which are output. Although we did not do it here, we can also have several input layers and/or several output layers, each with its own loss. This is useful when, for example, you want to create a network that takes as input an picture and a sound, and outputs the location of the objects in the picture that does that sound, along with a textual description of what is going on.

We now create, compile and fit the model as usual.

```
model = keras_model(input, output)

model %>% compile(
  # TODO use the mean squared error loss and the adam optimizer
)

model %>% fit(
  # TODO fit for two epochs on the training dataset
)
```

Let us check the reconstruction of a digit:

```
reconstruction = model %>% predict(x_train[2,,drop=FALSE])
reconstruction = pmin(1, pmax(reconstruction, 0))

dim(reconstruction) <- c(28, 28)

grid::grid.raster(reconstruction, interpolate = FALSE)
```

It is already quite good. Now try to remove the division of the KL by 784, train again and visualize the result.

You should see a gray blob that looks a bit like the average of many digits. This phenomenon is named *mode collapse*, i.e. the distribution of the generator collapsed to a single mode that covers the entire dataset, instead of (at least) one mode for every digit. In VAEs, this is typically caused by a KL term that is very strong at the beginning of training, and dominates the reconstruction loss. The optimizer will focus most of its efforts to reduce this term, ending up in a poor local minimum.

A popular method to deal with this issue is *KL annealing*. It consists in training the network without the KL regularizer for some time, then slowly increasing the weight of the KL. This procedure allows the network to first learn how to perform good reconstructions, then to adjust the latent code to conform to a Normal distribution without erasing progress on the reconstruction.

In order to implement this method, we need to change the variable `kl_weight` during training. It is possible to do this using a custom Keras callback. In case you do not remember, a callback provides a specific functionality at certain stages of the training process. In some previous labs, we used the early stopping callback to prevent overfitting.

Creating a custom callback is very similar to creating a custom layer (see the documentation here):

```
KlAnnealingCallback <- R6::R6Class("KlAnnealingCallback",
  inherit = KerasCallback,

  public = list(
    epoch_start = NULL,
    duration_epochs = NULL,
    variable = NULL,

    initialize = function(epoch_start, duration_epochs, variable) {
      self$epoch_start = epoch_start
      self$duration_epochs = duration_epochs
      self$variable = variable
    },

    # no kl at the beginning of training
    on_train_begin = function(logs = list()) {
      self$variable$assign(0.0)
    },

    # update the weight at the beginning of every epoch
    on_epoch_begin = function(epoch, logs = list()) {
      epoch = epoch + 1  # it starts from zero
      if(epoch < self$epoch_start) {
        message("epoch ", epoch, " is lower than starting epoch ", self$epoch_start)
      }
      else {
        new_weight = (
          # TODO linearly interpolate the new weight
        )

        self$variable$assign(new_weight)
        message("kl weight is ", new_weight, " at epoch ", epoch)
      }
    }
))

# TODO create the VAE as before

model = keras_model(input, output)

# here we isolate the encoder into its own model
# later we will use it for encoding samples
encoder = keras_model(input, latent_code)

model %>% compile(
  loss = 'mse',
  optimizer = 'adam',
)

model %>% fit(
  x_train, x_train, epochs = 20,
  callbacks = list(
    KlAnnealingCallback$new(epoch_start = 4, duration_epochs = 8, variable = kl_weight)
  )
)
```

```
reconstruction = model %>% predict(x_train[2,,drop=FALSE])
reconstruction = pmin(1, pmax(reconstruction, 0))
```

```r
dim(reconstruction) <- c(28, 28)

grid::grid.raster(reconstruction, interpolate = FALSE)
```