Lab 10

Exercise 1

In this exercise, we are going to implement an autoencoder and train it on the MNIST dataset. We will do this in Tensorflow, the library underlying Keras.

Tensorflow is much more general than Keras, in the sense that it allows you to write arbitrary computations, not limited to neural networks. It also provides *automatic differentiation*, which means that it can automatically compute gradients of every operation with respect to every variable. In practice, this means that any optimization problem solvable using analytic differentiation can be coded in Tensorflow. It also handles running the code on GPU, and/or distributing it across a cluster of machines.

You should have installed Tensorflow together with Keras, if not refer to this guide here.

```
library(tensorflow)
library(keras)

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

As an introduction, we will train PCA, i.e. an autoencoder with linear encoder $\mathbf{z} = \mathbf{E}\mathbf{x}$ and linear decoder: $\mathbf{x'} = \mathbf{D}\mathbf{z}$.

The first step is to create variables for $\mathbf{E}$ and $\mathbf{D}$.

```
latent_size = 64

weights_encoder = tf$Variable(
  matrix(
    rnorm(input_size * latent_size,  sd=sqrt(6 / (latent_size + input_size))),
    nrow = input_size
  )
)

weights_decoder = tf$Variable(
  matrix(
    rnorm(input_size * latent_size,  sd=sqrt(6 / (latent_size + input_size))),
    nrow = latent_size
  )
)
```

These two variables managed by Tensorflow, and are called *tensors*:

```
weights_encoder
```

Tensorflow can only operate on data that is stored inside a `tf$Variable`. If you do not create one yourself, Tensorflow will do it automatically, but this will considerably slow down your program.

Now, we can train the autoencoder for one epoch.

```
optimizer = tf$optimizers$Adam(learning_rate = 0.001)

batch_size = 32
num_batches = floor(nrow(x_train) / batch_size)

# tensorflow is very sensitive about the data types of variables:
# all operands must be of the same type, and you have to take care
# of this. the default data type is tf$float64.
# try to comment the following line and get familiar with the error message
# when you try to run this chunk of code, because you will encounter it often.
bs = tf$cast(batch_size, tf$float64)

for(i in 1:num_batches) {
  batch = x_train[((i - 1) * batch_size):(i * batch_size - 1),]

  # gradient tapes keep track of the operations that are performed on tensors
  with(tf$GradientTape() %as% tape, {
    # all the computations need to be performed inside this block,
    # otherwise the tape will not be able to compute the gradients

    # here we perform the forward pass, two simple matrix multiplications
    latent = tf$matmul(batch, weights_encoder)
    reconstruction = tf$matmul(latent, weights_decoder)

    # then compute the mean squared error
    loss = tf$reduce_sum(tf$square(reconstruction - batch)) / bs
  })

  # now we use the gradient tape to compute the gradient of the loss
  # with respect to the two matrices
  variables = list(weights_encoder, weights_decoder)
  gradients = tape$gradient(loss, variables)

  # finally, we pass the gradients and the respective variables to the optimizer
  # which will update the variables by performing one step of gradient descent
  optimizer$apply_gradients(list(
    list(gradients[[1]], variables[[1]]),
    list(gradients[[2]], variables[[2]])
  ))

  # we print the loss every once in a while
  if(i %% 250 == 0) {
    cat("Batch", i, "- Loss:", loss$numpy(), "\n")
    flush.console()
  }
}
```

This is plain Tensorflow, it is slow and nobody writes code like this anymore, but all the fancy things that Tensorflow offer eventually boil down to something like this.

Underlying Tensorflow there is a computational graph whose nodes are variables and operations. In newer versions of Tensorflow, this graph can be created automatically based on the code you write, so, in practice, it will not look too different from the chunk above. In this chunk, Tensorflow executes one operation at a time, and returns the result to R, which then calls the next Tensorflow operation. In the next chunks, Tensorflow

itself will execute all operations in sequence, running much faster.

Another new functionality of Tensorflow consists of utilities to load and transform data. This is especially convenient when your dataset does not fit in memory, and you have to load it from disk and transform it on the fly as training progresses.

```
if(!require(tfdatasets)) {
  install.packages("tfdatasets")
  library(tfdatasets)
}

batch_size = 32
num_epochs = 5

train_dataset = mnist$train$x %>%
  # with this function, we create a "dataset" from matrices that we have in memory
  # in particular, `mnist$train$x` is a 3D tensor of shape (60000, 28, 28)
  # this function creates samples by moving along the first dimension
  # i.e. the i-th sample is `mnist$train$x[i,,]`
  tensor_slices_dataset() %>%

  # here we transform every element of the dataset
  dataset_map(function(x) {
    # we first flatten it to a vector
    x = tf$reshape(x, c(1L, 28L * 28L))

    # then convert it to smaller precision
    x = tf$cast(x, tf$float32)

    # finally we normalize from 0 to 1
    x / 255
  }) %>%

  # shuffle the data at the beginning of every epoch
  dataset_shuffle(nrow(mnist$train$x)) %>%

  # iterate over the dataset for a given number of epochs
  dataset_repeat(num_epochs) %>%

  # split the dataset into batches of the given size
  dataset_batch(batch_size)
```

Now, we can use a simple for-loop to iterate over this dataset, and we will receive batches with the right size and the transformations we requested.

Before doing this, let us define more elaborate encoders and decoders. Try to have several dense layers and experiment with different activation functions (e.g. `tf$nn$sigmoid`, `tf$nn$relu`, etc.).

```
# an utility function to create the weights and biases for a dense layer
make_dense = function(input_size, output_size) {
  weights = matrix(
    rnorm(input_size * output_size,  sd=sqrt(6 / (output_size + input_size))),
    nrow = input_size
  )

  bias = matrix(rep(0, output_size), ncol = output_size)
```

```
  c(
    tf$Variable(weights, dtype = tf$float32),
    tf$Variable(bias, dtype = tf$float32)
  )
}

encoder_params = c(
  #!hwbegin TODO create the parameters needed for the encoder
  make_dense(784, 256),
  make_dense(256, 128),
  make_dense(128, 64)
  #!hwend
)

decoder_params = c(
  #!hwbegin TODO create the parameters needed for the decoder
  make_dense(64, 128),
  make_dense(128, 256),
  make_dense(256, 784)
  #!hwend
)

encoder = function(inputs) {
  #!hwbegin TODO encode the inputs
  x = tf$matmul(inputs, encoder_params[[1]]) + encoder_params[[2]]
  x = tf$nn$relu(x)
  x = tf$matmul(x, encoder_params[[3]]) + encoder_params[[4]]
  x = tf$nn$relu(x)
  x = tf$matmul(x, encoder_params[[5]]) + encoder_params[[6]]
  #!hwend
}

decoder = function(latent) {
  #!hwbegin TODO decode the latent variables
  x = tf$matmul(latent, decoder_params[[1]]) + decoder_params[[2]]
  x = tf$nn$relu(x)
  x = tf$matmul(x, decoder_params[[3]]) + decoder_params[[4]]
  x = tf$nn$relu(x)
  x = tf$matmul(x, decoder_params[[5]]) + decoder_params[[6]]
  #!hwend
}

# this library automatically creates the computational graph from the R code
if(!require(tfautograph)) {
  install.packages("tfautograph")
  library(tfautograph)
}

optimizer = tf$optimizers$Adam(learning_rate = 0.001)

training_step = autograph(function(batch) {
  #!hwbegin TODO perform the forward pass, computing the loss.\n# remember to use the gradient tape!
  with(tf$GradientTape() %as% tape, {
```

```
    reconstruction = decoder(encoder(batch))
    loss = tf$reduce_sum(tf$square(reconstruction - batch)) / as.numeric(nrow(batch))
  })
  #!hwend

  #!hwbegin TODO compute the gradients
  variables = c(encoder_params, decoder_params)
  gradients = tape$gradient(loss, variables)
  #!hwend

  #!hwbegin TODO do one step of gradient descent
  optimizer$apply_gradients(purrr::transpose(list(gradients, variables)))
  #!hwend

  loss
})


# simply wrap the top level function inside autograph to
# automatically generate the computational graph
train = autograph(function(num_batches) {
  i = 0
  for(batch in train_dataset) {
    loss = training_step(batch)

    if(i %% 250 == 0) {
      cat("Batch", i, "- Loss:", loss$numpy(), "\n")
      flush.console()
    }

    i = i + 1
    if(i >= num_batches) {
      break
    }
  }
})


train(1000)
```

Exercise 2

Consider a neural network with one hidden layer with threshold activation function (i.e., $\sigma(x) = \mathbf{1}[x > 0]$) and one output neuron with linear activation. Prove that such a neural network cannot *exactly* separate the cyan ($y = 1$) and white ($y = 0$) areas in Figure 1.

Note that the universal approximation theorem states that we can get *arbitrarily close* to this figure, it does not say that we can reconstruct it *exactly*.

Hint: Assume such a neural network exist, then consider one point in each of the four regions, and compute the difference between the predictions of points with different labels. You should reach a contradiction. You may assume a theorem stating that the hidden layer of such a neural network must contain two neurons corresponding to the two lines, and that no other lines pass through P.

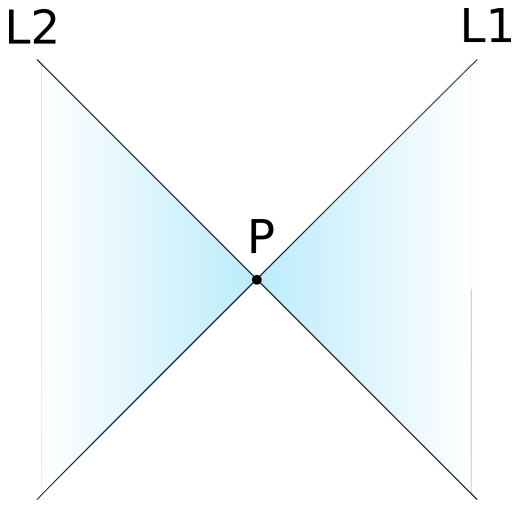

Figure 1: A region that cannot be classified correctly by a neural network with a single hidden layer.

Solution

Assume that such a neural network exists. Then, it computes a function g of the form:

$$g(\mathbf{x}) = \sum_{i=1}^{m} w_i \cdot \sigma(\mathbf{x}^T \mathbf{v}_i + c_i) + b$$

Now consider four points as in Figure 2, so that $g(\mathbf{x}_1) = g(\mathbf{x}_3) = 1$ and $g(\mathbf{x}_2) = g(\mathbf{x}_4) = 0$. Moreover, assume that the first hidden neuron models L_1, so that it has a positive activation for $\mathbf{x}_2$ and $\mathbf{x}_3$, and zero for $\mathbf{x}_1$ and $\mathbf{x}_4$.

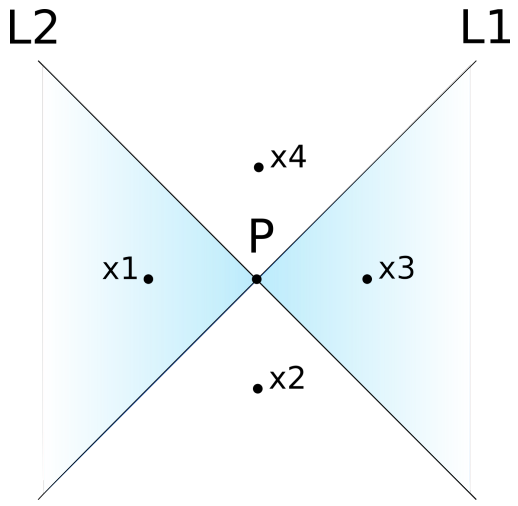

Figure 2: Four points in the four regions.

The difference between the predictions for $\mathbf{x}_1$ and $\mathbf{x}_2$ is

$$
\begin{aligned}
g(\mathbf{x}_1) - g(\mathbf{x}_2) &= \sum_{i=1}^{m} w_i \cdot \sigma(\mathbf{x}_1^T \mathbf{v}_i + c_i) + b - \sum_{i=1}^{m} w_i \cdot \sigma(\mathbf{x}_2^T \mathbf{v}_i + c_i) - b \\
&= w_1 \cdot \sigma(\mathbf{x}_1^T \mathbf{v}_1 + c_1) - w_1 \cdot \sigma(\mathbf{x}_2^T \mathbf{v}_1 + c_1) \\
&= w_1 \cdot 0 - w_1 \cdot 1 \\
&= -w_1 = 1
\end{aligned}
$$

The second step follows because when going from $\mathbf{x}_1$ to $\mathbf{x}_2$ we only cross L_1, and we stay on the same side of every other line. The last step follows because we know that $g(\mathbf{x}_1) = 1$ and that $g(\mathbf{x}_2) = 0$. This allows us to conclude that $w_1 = -1$.

The same reasoning applies to $\mathbf{x}_4$ and $\mathbf{x}_3$, too:

$$
\begin{aligned}
g(\mathbf{x}_4) - g(\mathbf{x}_3) &= w_1 \cdot \sigma(\mathbf{x}_4^T \mathbf{v}_1 + c_1) - w_1 \cdot \sigma(\mathbf{x}_3^T \mathbf{v}_1 + c_1)) \\
&= -w_1 = -1
\end{aligned}
$$

Which implies that $w_1 = 1$. This contradicts what we found previously, hence no such neural network exists.

Note: this proof, along with the theorems we assumed to be true but did not prove, are presented in (Blum and Li 1991).

Bibliography

Blum, Edward K., and Leong Kwan Li. 1991. "Approximation Theory and Feedforward Networks." *Neural Networks* 4 (4): 511–15. https://doi.org/10.1016/0893-6080(91)90047-9.