Resolución de un problema de clasificación con aprendizaje profundo utilizando un subconjunto del conjunto MNIST

Carlos García Gutiérrez (UO139393)

Introducción

La ejecución de esta práctica consta de las siguientes partes:
- Cargar en memoria los datos a utilizar
- Modelar y entrenar varias configuraciones de redes densas - Añadir regularización a las redes anteriores y comparar los resultados - Modelar y entrenar varias configuraciones de redes convolucionales

Carga de datos en memoria

```
library(keras)
library(ggplot2)
library(tidyr)
```

Obtenemos el dataset MNIST

```
mnist <- dataset_mnist()
```

Definimos una semilla con los dígitos del DNI y generamos una secuencia aleatoria con un tamaño de la mitad del de la lista de imágenes/etiquetas

```
set.seed(53540153)
sample_array <- sample.int(nrow(mnist$train$x), size = floor(.50 * nrow(mnist$train$x)))
```

Obtenemos la mitad de las imágenes/etiquetas para entrenar; el conjunto de test es el completo

```
train_images <- mnist$train$x[sample_array,,]
train_labels <- mnist$train$y[sample_array]
test_images <- mnist$test$x
test_labels <- mnist$test$y
```

Se reordenan los datos para poder ser usados como entrada de las redes densas y se escalan los valores RGB de las imágenes para que estén en el intervalo [0, 1], asimismo se transforman las etiquetas a valores binarios, según el dígito que representen

```
train_images <- array_reshape(train_images, c(nrow(train_images), 28 * 28))
train_images <- train_images / 255
test_images <- array_reshape(test_images, c(nrow(test_images), 28 * 28))
test_images <- test_images / 255
train_labels <- to_categorical(train_labels)
test_labels <- to_categorical(test_labels)
```

Redes neuronales densas

Vamos a crear tres redes neuronales densas: con dos capas (entrada y salida), con tres capas (igual que la anterior pero añadiendo una capa oculta) y con cuatros capas (igual que la primera pero añadiendo dos capas ocultas)

Las capa de entrada contiene una neurona por cada pixel (28 x 28) y estas se activan utilizando la función ReLU, que es la adecuada para la escala de grises de las imágenes

La capa de salida tiene 10 neuronas, que son las necesarias para las 10 categorías a considerar (valores del 0 al 9), en este caso la función de activación es la adecuada para problemas de clasificación múltiple de una etiqueta (como es nuestro caso)

Evidentemente, para un problema tan sencillo, no serían necesarias redes con tantas capas, pero se ha decidido utilizar estas configuraciones para ilustrar el problema del sobreajuste e intentar minimizarlo posteriormente mediante la regularización

Para todas las redes, la función a minimizar es "categorical_crossentropy", que es la adecuada para problemas de clasificación múltiple de una etiqueta. La optimización estará basada en el descenso del gradiente utilizando solo un conjunto de los pesos, según lo visto en clase

```
dense_network_2layers <- keras_model_sequential() %>%
  layer_dense(units = 512, activation = "relu", input_shape = c(28 * 28)) %>%
  layer_dense(units = 10, activation = "softmax")

dense_network_2layers %>% compile(
  optimizer = "rmsprop",
  loss = "categorical_crossentropy",
  metrics = c("accuracy")
)

summary(dense_network_2layers)
```

```
## _________________________________________________________________________
## Layer (type)                     Output Shape                  Param #
## =========================================================================
## dense (Dense)                    (None, 512)                   401920
## _________________________________________________________________________
## dense_1 (Dense)                  (None, 10)                    5130
## =========================================================================
## Total params: 407,050
## Trainable params: 407,050
## Non-trainable params: 0
## _________________________________________________________________________
```

```
dense_network_3layers <- keras_model_sequential() %>%
  layer_dense(units = 512, activation = "relu", input_shape = c(28 * 28)) %>%
  layer_dense(units = 512, activation = "relu") %>%
  layer_dense(units = 10, activation = "softmax")

dense_network_3layers %>% compile(
  optimizer = "rmsprop",
  loss = "categorical_crossentropy",
  metrics = c("accuracy")
)

summary(dense_network_3layers)
```

```
## _________________________________________________________________
## Layer (type)                 Output Shape              Param #
## =================================================================
## dense_2 (Dense)              (None, 512)               401920
## _________________________________________________________________
## dense_3 (Dense)              (None, 512)               262656
## _________________________________________________________________
## dense_4 (Dense)              (None, 10)                5130
## =================================================================
## Total params: 669,706
## Trainable params: 669,706
## Non-trainable params: 0
## _________________________________________________________________
```

```
dense_network_4layers <- keras_model_sequential() %>%
  layer_dense(units = 512, activation = "relu", input_shape = c(28 * 28)) %>%
  layer_dense(units = 512, activation = "relu") %>%
  layer_dense(units = 512, activation = "relu") %>%
  layer_dense(units = 10, activation = "softmax")

dense_network_4layers %>% compile(
  optimizer = "rmsprop",
  loss = "categorical_crossentropy",
  metrics = c("accuracy")
)

summary(dense_network_4layers)
```

```
## _________________________________________________________________
## Layer (type)                 Output Shape              Param #
## =================================================================
## dense_5 (Dense)              (None, 512)               401920
## _________________________________________________________________
## dense_6 (Dense)              (None, 512)               262656
## _________________________________________________________________
## dense_7 (Dense)              (None, 512)               262656
## _________________________________________________________________
## dense_8 (Dense)              (None, 10)                5130
## =================================================================
## Total params: 932,362
## Trainable params: 932,362
## Non-trainable params: 0
## _________________________________________________________________
```

Se realiza el entrenamiento de las redes (utilizando cinco iteraciones)

```
dense_network_2layers %>% fit(train_images, train_labels, epochs = 5, batch_size = 128)
dense_network_3layers %>% fit(train_images, train_labels, epochs = 5, batch_size = 128)
dense_network_4layers %>% fit(train_images, train_labels, epochs = 5, batch_size = 128)
```

Se obtienen y se muestran los resultados

```
metrics_dn_2layers <- dense_network_2layers %>% evaluate(test_images, test_labels)
metrics_dn_3layers <- dense_network_3layers %>% evaluate(test_images, test_labels)
metrics_dn_4layers <- dense_network_4layers %>% evaluate(test_images, test_labels)
metrics_dn_2layers
```

```
## $loss
## [1] 0.1061436
##
## $acc
## [1] 0.9672
```

```
metrics_dn_3layers
```

```
## $loss
## [1] 0.1193921
##
## $acc
## [1] 0.9687
```

```
metrics_dn_4layers
```

```
## $loss
## [1] 0.1707906
##
## $acc
## [1] 0.9602
```

Se puede observar que añadir una capa oculta a la red mejora los resultados, pero añadir una segunda capa oculta los vuelve a empeorar; esto era de esperar ya que las redes más profundas producen un sobreajuste al modelo, podrías decir entonces, que una configuración con una capa oculta es la que mejor se ajusta a este problema y este conjunto de datos.

Se podría aunmentar también el sobreajuste haciendo las capas ocultas más grandes o añadiendo más iteraciones, pero con las tres redes utilizadas queda conseguido perfectamente el objetivo de ilustrar como una red compleja tiende a sobreajustar.

Regularización

La regularización es una técnica que intenta mitigar el sobreajuste de las redes neuronales basándose en el principio de que, a igualdad de condiciones, se debe utilizar el modelo más sencillo. Para ello se intenta limitar la complejidad de la red. Una estrategia consiste en obligar a sus pesos a tomar valores pequeños, mediante la introducción de una penalización para los valores altos. Otra estrategia consiste eliminar neuronas durante el entrenamiento, con la idea de que la introducción de ruido en la salida de una capa puede hacer que la red ignore los patrones menos significativos.

Para la regularización, vamos a utilizar la red de cuatro capas que era la que mayor sobreajuste presentaba y será la que mjor nos sirva para ilustrar la regularización.

Empezamos añadiendo regularización de la norma L1 de los pesos

```
dense_network_4layers_regL1 <- keras_model_sequential() %>%
  layer_dense(units = 512, kernel_regularizer = regularizer_l1(0.001), activation = "relu", input_shape
  layer_dense(units = 512, kernel_regularizer = regularizer_l1(0.001), activation = "relu") %>%
```

```
  layer_dense(units = 512, kernel_regularizer = regularizer_l1(0.001), activation = "relu") %>%
  layer_dense(units = 10, activation = "softmax")

dense_network_4layers_regL1 %>% compile(
  optimizer = "rmsprop",
  loss = "categorical_crossentropy",
  metrics = c("accuracy")
)

summary(dense_network_4layers_regL1)
```

```
## _________________________________________________________________________
## Layer (type)                    Output Shape                  Param #
## =========================================================================
## dense_9 (Dense)                 (None, 512)                   401920
## _________________________________________________________________________
## dense_10 (Dense)                (None, 512)                   262656
## _________________________________________________________________________
## dense_11 (Dense)                (None, 512)                   262656
## _________________________________________________________________________
## dense_12 (Dense)                (None, 10)                    5130
## =========================================================================
## Total params: 932,362
## Trainable params: 932,362
## Non-trainable params: 0
## _________________________________________________________________________
```

Seguimos con regularización de la norma L2 de los pesos

```
dense_network_4layers_regL2 <- keras_model_sequential() %>%
  layer_dense(units = 512, kernel_regularizer = regularizer_l2(0.001), activation = "relu", input_shape
  layer_dense(units = 512, kernel_regularizer = regularizer_l2(0.001), activation = "relu") %>%
  layer_dense(units = 512, kernel_regularizer = regularizer_l2(0.001), activation = "relu") %>%
  layer_dense(units = 10, activation = "softmax")

dense_network_4layers_regL2 %>% compile(
  optimizer = "rmsprop",
  loss = "categorical_crossentropy",
  metrics = c("accuracy")
)

summary(dense_network_4layers_regL2)
```

```
## _________________________________________________________________________
## Layer (type)                    Output Shape                  Param #
## =========================================================================
## dense_13 (Dense)                (None, 512)                   401920
## _________________________________________________________________________
## dense_14 (Dense)                (None, 512)                   262656
## _________________________________________________________________________
## dense_15 (Dense)                (None, 512)                   262656
## _________________________________________________________________________
```

```
## dense_16 (Dense)                 (None, 10)                    5130
## ================================================================================
## Total params: 932,362
## Trainable params: 932,362
## Non-trainable params: 0
## ________________________________________________________________________________
```

Finalmente utilizamos un dropout (del 50% de cada capa)

```
dense_network_4layers_dropout <- keras_model_sequential() %>%
  layer_dense(units = 512, activation = "relu", input_shape = c(28 * 28)) %>%
  layer_dropout(rate = 0.5) %>%
  layer_dense(units = 512, activation = "relu") %>%
  layer_dropout(rate = 0.5) %>%
  layer_dense(units = 512, activation = "relu") %>%
  layer_dropout(rate = 0.5) %>%
  layer_dense(units = 10, activation = "softmax")

dense_network_4layers_dropout %>% compile(
  optimizer = "rmsprop",
  loss = "categorical_crossentropy",
  metrics = c("accuracy")
)

summary(dense_network_4layers_dropout)
```

```
## ________________________________________________________________________________
## Layer (type)                     Output Shape                  Param #
## ================================================================================
## dense_17 (Dense)                 (None, 512)                   401920
## ________________________________________________________________________________
## dropout (Dropout)                (None, 512)                   0
## ________________________________________________________________________________
## dense_18 (Dense)                 (None, 512)                   262656
## ________________________________________________________________________________
## dropout_1 (Dropout)              (None, 512)                   0
## ________________________________________________________________________________
## dense_19 (Dense)                 (None, 512)                   262656
## ________________________________________________________________________________
## dropout_2 (Dropout)              (None, 512)                   0
## ________________________________________________________________________________
## dense_20 (Dense)                 (None, 10)                    5130
## ================================================================================
## Total params: 932,362
## Trainable params: 932,362
## Non-trainable params: 0
## ________________________________________________________________________________
```

Antes de continuar, vamos a entrenar otras 5 iteraciones a la red neuronal de cuatro capas, para para forzar aún más el sobreajuste y poder comparar mejor los efectos de la regularización

```
dense_network_4layers_hist <- dense_network_4layers %>%
  fit(train_images, train_labels, epochs = 10, batch_size = 128, initial_epoch = 5)
```

Se realiza el entrenamiento de las redes (utilizando 10 iteraciones)

```
dense_network_4layers_regL1_hist <- dense_network_4layers_regL1 %>%
  fit(train_images, train_labels, epochs = 10, batch_size = 128)
dense_network_4layers_regL2_hist <- dense_network_4layers_regL2 %>%
  fit(train_images, train_labels, epochs = 10, batch_size = 128)
dense_network_4layers_dropout_hist <- dense_network_4layers_dropout %>%
  fit(train_images, train_labels, epochs = 10, batch_size = 128)
```

Se obtienen y se muestran los resultados

```
metrics_dn_4layers <- dense_network_4layers %>% evaluate(test_images, test_labels)
metrics_dn_4layers_L1 <- dense_network_4layers_regL1 %>% evaluate(test_images, test_labels)
metrics_dn_4layers_L2 <- dense_network_4layers_regL2 %>% evaluate(test_images, test_labels)
metrics_dn_4layers_dropout <- dense_network_4layers_dropout %>% evaluate(test_images, test_labels)
metrics_dn_4layers
```

```
## $loss
## [1] 0.1829455
##
## $acc
## [1] 0.9712
```

```
metrics_dn_4layers_L1
```

```
## $loss
## [1] 0.9722705
##
## $acc
## [1] 0.9249
```

```
metrics_dn_4layers_L2
```

```
## $loss
## [1] 0.184392
##
## $acc
## [1] 0.9699
```

```
metrics_dn_4layers_dropout
```

```
## $loss
## [1] 0.1210617
##
## $acc
## [1] 0.9736
```

Como se puede observar en los resultados el sobreajuste únicamente mejora con la regularización que utiliza "dropout", pero no con las de las normas de los pesos. Una interpretación plausible podría ser que los pesos de la red ya eran pequeños antes de introducir la regularización y que, por otro lado, existe ruido en los los datos de origen y, gracias al "dropout", se evita que el modelo sobreajuste al no tener en cuenta dicho ruido.

NOTA: lo ideal hubisese sido mostrar una gráfica con la evulucion de la función de pérdida a lo largo de las iteraciones, para cada uno de los cuatros modelos, en vez de mostrar en texto los resultados finales de la última iteración. Sin embargo, no me ha sido posible mostar ningún gráfico en mi ordenador, ya que cada
vez que se intentaba lanzar uno RStudio empezaba a consumir memoria y tras un par de minutos acababa quedándose
bloqueado, haciendo necesario finalizar el proceso manualmente. No he sido capaz de solucionar dicho problema.

Redes convolucionales

Empezamos creando las redes convolucionales. Debemos asegurarnos de que reciba como entrada tensores de
tamaño 28x28x1 y de que la salida tenga 10 neuronas (una por cada categoría/dígito). El tamaño del "pooling", de los "kernels" y el número de filtros varían entre cada uno de los ejemplos.

```
conv_network_a <- keras_model_sequential() %>%
  layer_conv_2d(filters = 32, kernel_size = c(2, 2), activation = "relu", input_shape = c(28, 28, 1)) %
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 32, kernel_size = c(2, 2), activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 32, kernel_size = c(2, 2), activation = "relu") %>%
  layer_flatten() %>%
  layer_dense(units = 32, activation = "relu") %>%
  layer_dense(units = 10, activation = "softmax")

conv_network_a %>% compile(
  optimizer = "rmsprop",
  loss = "categorical_crossentropy",
  metrics = c("accuracy")
)

summary(conv_network_a)
```

```
## _________________________________________________________________
## Layer (type)                 Output Shape              Param #
## =================================================================
## conv2d (Conv2D)              (None, 27, 27, 32)        160
## _________________________________________________________________
## max_pooling2d (MaxPooling2D) (None, 13, 13, 32)        0
## _________________________________________________________________
## conv2d_1 (Conv2D)            (None, 12, 12, 32)        4128
## _________________________________________________________________
## max_pooling2d_1 (MaxPooling2D) (None, 6, 6, 32)        0
## _________________________________________________________________
## conv2d_2 (Conv2D)            (None, 5, 5, 32)          4128
## _________________________________________________________________
## flatten (Flatten)            (None, 800)               0
## _________________________________________________________________
```

```
## dense_21 (Dense)                 (None, 32)                    25632
## ________________________________________________________________________
## dense_22 (Dense)                 (None, 10)                    330
## ========================================================================
## Total params: 34,378
## Trainable params: 34,378
## Non-trainable params: 0
## ________________________________________________________________________
```

```
conv_network_b <- keras_model_sequential() %>%
  layer_conv_2d(filters = 32, kernel_size = c(2, 2), activation = "relu", input_shape = c(28, 28, 1)) %
  layer_max_pooling_2d(pool_size = c(3, 3)) %>%
  layer_conv_2d(filters = 64, kernel_size = c(2, 2), activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(3, 3)) %>%
  layer_conv_2d(filters = 64, kernel_size = c(2, 2), activation = "relu") %>%
  layer_flatten() %>%
  layer_dense(units = 64, activation = "relu") %>%
  layer_dense(units = 10, activation = "softmax")

conv_network_b %>% compile(
  optimizer = "rmsprop",
  loss = "categorical_crossentropy",
  metrics = c("accuracy")
)

summary(conv_network_b)
```

```
## ________________________________________________________________________
## Layer (type)                     Output Shape                  Param #
## ========================================================================
## conv2d_3 (Conv2D)                (None, 27, 27, 32)            160
## ________________________________________________________________________
## max_pooling2d_2 (MaxPooling2D)   (None, 9, 9, 32)              0
## ________________________________________________________________________
## conv2d_4 (Conv2D)                (None, 8, 8, 64)              8256
## ________________________________________________________________________
## max_pooling2d_3 (MaxPooling2D)   (None, 2, 2, 64)              0
## ________________________________________________________________________
## conv2d_5 (Conv2D)                (None, 1, 1, 64)              16448
## ________________________________________________________________________
## flatten_1 (Flatten)              (None, 64)                    0
## ________________________________________________________________________
## dense_23 (Dense)                 (None, 64)                    4160
## ________________________________________________________________________
## dense_24 (Dense)                 (None, 10)                    650
## ========================================================================
## Total params: 29,674
## Trainable params: 29,674
## Non-trainable params: 0
## ________________________________________________________________________
```

```
conv_network_c <- keras_model_sequential() %>%
  layer_conv_2d(filters = 32, kernel_size = c(4, 4), activation = "relu", input_shape = c(28, 28, 1)) %
```

```
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 128, kernel_size = c(4, 4), activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 128, kernel_size = c(4, 4), activation = "relu") %>%
  layer_flatten() %>%
  layer_dense(units = 128, activation = "relu") %>%
  layer_dense(units = 10, activation = "softmax")

conv_network_c %>% compile(
  optimizer = "rmsprop",
  loss = "categorical_crossentropy",
  metrics = c("accuracy")
)

summary(conv_network_c)
```

```
## _________________________________________________________________________
## Layer (type)                 Output Shape              Param #
## =========================================================================
## conv2d_6 (Conv2D)            (None, 25, 25, 32)        544
## _________________________________________________________________________
## max_pooling2d_4 (MaxPooling2D)  (None, 12, 12, 32)     0
## _________________________________________________________________________
## conv2d_7 (Conv2D)            (None, 9, 9, 128)         65664
## _________________________________________________________________________
## max_pooling2d_5 (MaxPooling2D)  (None, 4, 4, 128)      0
## _________________________________________________________________________
## conv2d_8 (Conv2D)            (None, 1, 1, 128)         262272
## _________________________________________________________________________
## flatten_2 (Flatten)          (None, 128)               0
## _________________________________________________________________________
## dense_25 (Dense)             (None, 128)               16512
## _________________________________________________________________________
## dense_26 (Dense)             (None, 10)                1290
## =========================================================================
## Total params: 346,282
## Trainable params: 346,282
## Non-trainable params: 0
## _________________________________________________________________________
```

Obtenemos de nuevo las imágenes/etiquetas de entrenamiento y de test

```
train_images <- mnist$train$x[sample_array,,]
test_images <- mnist$test$x
```

Se reordenan los datos para poder ser usados como entrada de las redes convolucionales y se escalan los valores RGB de las imágenes para que estén en el intervalo [0, 1]

```
train_images <- array_reshape(train_images, c(nrow(train_images), 28, 28, 1))
train_images <- train_images / 255
test_images <- array_reshape(test_images, c(nrow(test_images), 28, 28, 1))
test_images <- test_images / 255
```

Se realiza el entrenamiento de las redes

```
conv_network_a %>% fit(train_images, train_labels, epochs = 5, batch_size = 64)
conv_network_b %>% fit(train_images, train_labels, epochs = 5, batch_size = 64)
conv_network_c %>% fit(train_images, train_labels, epochs = 5, batch_size = 64)
```

Se obtienen y se muestran los resultados

```
metrics_cn_a <- conv_network_a %>% evaluate(test_images, test_labels)
metrics_cn_b <- conv_network_b %>% evaluate(test_images, test_labels)
metrics_cn_c <- conv_network_c %>% evaluate(test_images, test_labels)

metrics_cn_a
```

```
## $loss
## [1] 0.05867318
##
## $acc
## [1] 0.9822
```

```
metrics_cn_b
```

```
## $loss
## [1] 0.1011054
##
## $acc
## [1] 0.9693
```

```
metrics_cn_c
```

```
## $loss
## [1] 0.03519123
##
## $acc
## [1] 0.991
```

Se puede observar como resultado que los peores resultados se obtienen en red convolucional B. Esto era esperable ya que la combinación del tamaño de kernel y de operadores de pooling no es la adecuada. Los mejores resultados se obtienen con la red convolucional C, probablemente debiso a su número de filtros, pero a costa de un notabilísimo coste computacional.

Partiremos ahora la red convolucional A para realizar las combinaciones de capas de de convolución y de pooling que se solicitan. Podría partirse también de la red convolucional C, y muy probablemente se obtendrían mejores resultados, pero el coste computacional sería elevadísimo. La red convolucional A es una elección más equilibrada.

Primera combinación: capa convolucional transpuesta 2D y capa de max_pooling 1D

```
#conv_network_a1 <- keras_model_sequential() %>%
#  layer_conv_2d_transpose(filters = 32, kernel_size = c(2, 2), activation = "relu", input_shape = c(28
#  layer_max_pooling_1d(pool_size = 2) %>%
#  layer_conv_2d_transpose(filters = 32, kernel_size = c(2, 2), activation = "relu") %>%
```

```
#  layer_max_pooling_1d(pool_size = 2) %>%
#  layer_conv_2d_transpose(filters = 32, kernel_size = c(2, 2), activation = "relu") %>%
#  layer_flatten() %>%
#  layer_dense(units = 32, activation = "relu") %>%
#  layer_dense(units = 10, activation = "softmax")
```

El resultado es que no se puede realizar pooling 1D sobre el resultado de una capaz de mayor dimensión

Segunda combinación: capa convolucional transpuesta 2D y capa de max_pooling 2D

```
conv_network_a2 <- keras_model_sequential() %>%
  layer_conv_2d_transpose(filters = 32, kernel_size = c(2, 2), activation = "relu", input_shape = c(28,
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d_transpose(filters = 32, kernel_size = c(2, 2), activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d_transpose(filters = 32, kernel_size = c(2, 2), activation = "relu") %>%
  layer_flatten() %>%
  layer_dense(units = 32, activation = "relu") %>%
  layer_dense(units = 10, activation = "softmax")

conv_network_a2 %>% compile(
  optimizer = "rmsprop",
  loss = "categorical_crossentropy",
  metrics = c("accuracy")
)

summary(conv_network_a2)
```

```
## _________________________________________________________________________
## Layer (type)                     Output Shape                  Param #
## =========================================================================
## conv2d_transpose (Conv2DTranspos (None, 29, 29, 32)            160
## _________________________________________________________________________
## max_pooling2d_6 (MaxPooling2D)   (None, 14, 14, 32)            0
## _________________________________________________________________________
## conv2d_transpose_1 (Conv2DTransp (None, 15, 15, 32)            4128
## _________________________________________________________________________
## max_pooling2d_7 (MaxPooling2D)   (None, 7, 7, 32)              0
## _________________________________________________________________________
## conv2d_transpose_2 (Conv2DTransp (None, 8, 8, 32)              4128
## _________________________________________________________________________
## flatten_3 (Flatten)              (None, 2048)                  0
## _________________________________________________________________________
## dense_27 (Dense)                 (None, 32)                    65568
## _________________________________________________________________________
## dense_28 (Dense)                 (None, 10)                    330
## =========================================================================
## Total params: 74,314
## Trainable params: 74,314
## Non-trainable params: 0
## _________________________________________________________________________
```

El resultado del entrenamiento y la evaluación se muestra a contionuación

```
conv_network_a2 %>% fit(train_images, train_labels, epochs = 5, batch_size = 64)
metrics_cn_a2 <- conv_network_c %>% evaluate(test_images, test_labels)
metrics_cn_a2
```

```
## $loss
## [1] 0.03519123
##
## $acc
## [1] 0.991
```

Tercera combinación: capa convolucional transpuesta 2D y capa de average_pooling 2D

```
conv_network_a3 <- keras_model_sequential() %>%
  layer_conv_2d_transpose(filters = 32, kernel_size = c(2, 2), activation = "relu", input_shape = c(28,
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d_transpose(filters = 32, kernel_size = c(2, 2), activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d_transpose(filters = 32, kernel_size = c(2, 2), activation = "relu") %>%
  layer_flatten() %>%
  layer_dense(units = 32, activation = "relu") %>%
  layer_dense(units = 10, activation = "softmax")

conv_network_a3 %>% compile(
  optimizer = "rmsprop",
  loss = "categorical_crossentropy",
  metrics = c("accuracy")
)

summary(conv_network_a3)
```

```
## _________________________________________________________________________
## Layer (type)                     Output Shape                  Param #
## =========================================================================
## conv2d_transpose_3 (Conv2DTransp (None, 29, 29, 32)            160
## _________________________________________________________________________
## max_pooling2d_8 (MaxPooling2D)   (None, 14, 14, 32)            0
## _________________________________________________________________________
## conv2d_transpose_4 (Conv2DTransp (None, 15, 15, 32)            4128
## _________________________________________________________________________
## max_pooling2d_9 (MaxPooling2D)   (None, 7, 7, 32)              0
## _________________________________________________________________________
## conv2d_transpose_5 (Conv2DTransp (None, 8, 8, 32)              4128
## _________________________________________________________________________
## flatten_4 (Flatten)              (None, 2048)                  0
## _________________________________________________________________________
## dense_29 (Dense)                 (None, 32)                    65568
## _________________________________________________________________________
## dense_30 (Dense)                 (None, 10)                    330
## =========================================================================
## Total params: 74,314
## Trainable params: 74,314
## Non-trainable params: 0
## _________________________________________________________________________
```

El resultado se muestra a continuación del entrenamiento y la evaluación se muestra a contionuación

```
conv_network_a3 %>% fit(train_images, train_labels, epochs = 5, batch_size = 64)
metrics_cn_a3 <- conv_network_c %>% evaluate(test_images, test_labels)
metrics_cn_a3
```

```
## $loss
## [1] 0.03519123
##
## $acc
## [1] 0.991
```

Cuarta combinación: capa convolucional 3D y capa de max_pooling 2D

```
#conv_network_a4 <- keras_model_sequential() %>%
#  layer_conv_3d(filters = 32, kernel_size = c(2, 2, 2), activation = "relu", input_shape = c(28, 28, 1
#  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
#  layer_conv_3d_transpose(filters = 32, kernel_size = c(2, 2, 2), activation = "relu") %>%
#  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
#  layer_conv_3d_transpose(filters = 32, kernel_size = c(2, 2, 2), activation = "relu") %>%
#  layer_flatten() %>%
#  layer_dense(units = 32, activation = "relu") %>%
#  layer_dense(units = 10, activation = "softmax")
```

El resultado es que la capa de entrada no puede ser de dimensión mayor que la dimensión de los datos

Quinta combinación: capa convolucional 2D y capa de global_max_pooling 2D

```
#conv_network_a5 <- keras_model_sequential() %>%
#  layer_conv_2d(filters = 32, kernel_size = c(2, 2), activation = "relu", input_shape = c(28, 28, 1)) %
#  layer_global_max_pooling_2d() %>%
#  layer_conv_2d(filters = 32, kernel_size = c(2, 2), activation = "relu") %>%
#  layer_global_max_pooling_2d() %>%
#  layer_conv_2d(filters = 32, kernel_size = c(2, 2), activation = "relu") %>%
#  layer_flatten() %>%
#  layer_dense(units = 32, activation = "relu") %>%
#  layer_dense(units = 10, activation = "softmax")
```

El resultado es que la capa pooling global devuelve una salida de dimensión menor a la necesaria por la siguiente capa convolucional