ML: Assignment 4

December 3, 2023

Time used for reading: 5
Time used for the basic assignment: 15
Time used for extra assignment (VG): 7
Good with lab:
One really good thing with this lab was that I got to work with hyperparameters in a convolution neural network when doing a random search on the neural network.

Things to improve in the lab:
No major area of improvement in this lab, however one would be to include a hint on which functions to use when visualize an image from the CIFAT10 dataset in subsection 1.7.

Anything that was difficult with the material?
The coding itself was not difficult or time consuming but the computational and downloading packages took some time. Hence, the total number of hours spend on this lab was mostly due to the time to run the code.

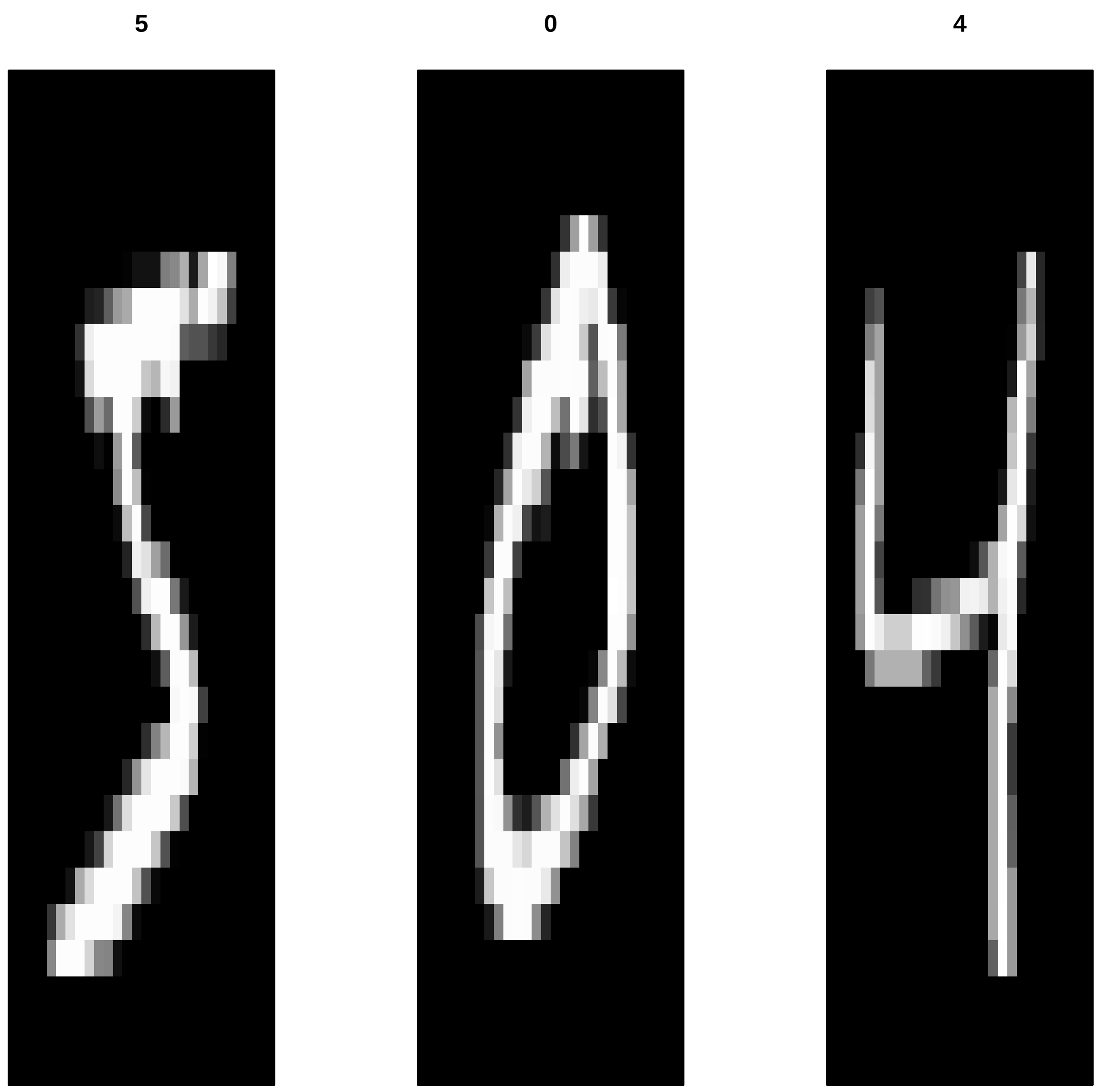
Task 1: Convolutional neural networks using Keras

```
setwd("~/Desktop/Machine Learning")
library(tensorflow)
library(keras)
mnist <- dataset_mnist()
x_train <- mnist$train$x <- mnist$train$x/255
x_test <- mnist$test$x <- mnist$test$x/255
y_train <- mnist$train$y
y_test <- mnist$test$y
```

1.1

As a first step we visualize and report the first three digits from the training dataset as an image.

```
par(mfrow = c(1,3))

for (idx in 1:3) {

  im <- x_train[idx,,]
  im <- t(apply(im, 2, rev))
  image(1:28, 1:28, im, col=gray((0:255)/255), xlab = "", ylab = "",
      xaxt='n', yaxt='n', main= paste(y_train[idx]))
}
```



The first, second and third handwritten digit in the training dataset is a 5, 0 and 4, respectively.

1.2

Now we implement a Convolutional Neural Network for the MNIST dataset and the network consists of two Convolutional layers.

```r
model <- keras_model_sequential() %>%

  layer_conv_2d(32, kernel_size = c(3,3),
                input_shape =  c(28, 28, 1), activation = "relu") %>%

  layer_max_pooling_2d(pool_size = c(2,2)) %>%

  layer_conv_2d(32, kernel_size = c(3,3),
                activation = 'relu') %>%

  layer_flatten() %>%
  layer_dense(units = 64, activation = 'relu') %>%
  layer_dense(units = 10, activation = 'softmax')

summary(model)
```

```
## Model: "sequential"
## ________________________________________________________________________________
##  Layer (type)                       Output Shape                    Param #
## ================================================================================
##  conv2d_1 (Conv2D)                  (None, 26, 26, 32)              320
##  max_pooling2d (MaxPooling2D)       (None, 13, 13, 32)              0
##  conv2d (Conv2D)                    (None, 11, 11, 32)              9248
##  flatten (Flatten)                  (None, 3872)                    0
##  dense_1 (Dense)                    (None, 64)                      247872
##  dense (Dense)                      (None, 10)                      650
## ================================================================================
## Total params: 258090 (1008.16 KB)
## Trainable params: 258090 (1008.16 KB)
## Non-trainable params: 0 (0.00 Byte)
## ________________________________________________________________________________
```

1.3

From the output in section 1.2 it shows that the first layer consists of 320 parameters and the total number of parameters in a layer is decided by the dimensions of the Kernel matrix K, the bias and the number of filters. In our model we have that K is a 3×3 matrix and 32 number of filters. We can calculate the number of parameters which in our case corresponds to the following:

$$T = ((3 \times 3) + 1) \times 32 = 320$$

where T denotes the total number of parameters for the layer. Further, of the 320 parameters we have that $(3 \times 3) \times 32) = 288$ are Kernel weights while $1 \times 32 = 32$ are biases.

1.4: Calculating the loss and the accurarcy

Now we will train the network created above, in section 1.2, by using Keras and looking at the loss and accuracy on the MNIST dataset. We are estimating the loss and the accuracy by using the compile and fit function in the keras package in R and when fitting the network we use 10 epochs.

```
loss_fn <- loss_sparse_categorical_crossentropy(from_logits = TRUE)

model %>% compile(
  optimizer = "adam",
  loss = loss_fn,
  metrics = "accuracy"
)
```

```
model %>% fit(
  x_train, y_train,
  epochs = 10, validation_split = 0.2)

1500/1500 [==============================] - 42s 28ms/step - loss: 0.0058 - accuracy: 0.9982 -
val_loss: 0.0601 - val_accuracy: 0.9866
```

When fitting the network, using 10 epochs, the loss and accuracy is estimated to 0.0058 and 0.9982, respectively, while the validation loss and validation accuracy is estimated to 0.0601 and 0.9866, respectively.

1.5

For this subsection we will choose three different hyperparameters to evaluate and we decide to use number of neurons, dropout rate as well as learning rate as the hyperparameters to evaluate. Since we want to fit 10 different models with random hyperparameters settings we will start to assign each parameter a statistical distribution. They have been assigned the following distributions for where we will take 10 random draws, one for each fitted model.

$hidden - unit \sim U(10, 90)$

$dropout - rate \sim U(0, 0.5)$

$log - learning - rate \sim U(-5, -1)$

where $U(\cdot)$ denotes the uniform distribution. Further we have that $learning - rate = 10^{log-learning-rate}$. Since the number of neurons need to be an integer in the layer dense function we decide to round the drawn sample to the closet integer, similar is being done to the log learning rate hyperparameters. This is however due to convenience reasons when transforming from log learning rate to the learning rate hyperparameter.

```
# the hyperparameters
# number of neurons in the layer_dence function
hidden_units <- round( runif(n = 10, min =  10, max = 90) )
# the dropout rate for the layer_dropout function
dropout <- runif(n = 10, min =  0, max =  0.5)
# learning rate for the compile and optimizer_rmsprop() function
learningrate <- 10^(round ( runif(n =10, min =  -5, max = -1)))

random_search <- function(hidden_units, dropout, learningrate) {
  best_model <- NULL; best_hyperparaeter <- NULL; best_val_accuracy <- 0

  # which corresponds to 10 since we want to fit 10 different models
  # and we will index the drawn sample for the hyperparameters
  for (i in 1:length(hidden_units) ) {

  # for this part we create a keras model with the same structure, as before in
  # section 1.4. However, we will add argument of learning rate, dropout rate and
  # the number of hidden units. This will be done as a function argument in
  # layer_dense funciton for hidden units hyperparmaters, compile and optimizer_rmsprop
```

```r
# function for learning rate as well as layer_dropout function for drop out rate.

# we will base this on the MNIST dataset, training dataset
# and we set relu as the activation function for each convoluational
# layer except for the output layer which got the softmax activation function.
model <- keras_model_sequential() %>%
layer_conv_2d(32, kernel_size = c(3,3),
              activation = "relu", input_shape = c(28,28,1))%>%
layer_max_pooling_2d(pool_size = c(2, 2)) %>%
layer_dropout(dropout[i]) %>%

layer_conv_2d(32, kernel_size = c(3,3),
              activation = 'relu') %>%
layer_dropout(dropout[i]) %>%

layer_flatten() %>%
layer_dense(units = hidden_units[i],
            activation = "relu") %>%

layer_dense(units = nlevels(as.factor(y_train)),
            activation = 'softmax')

loss_fn <- loss_sparse_categorical_crossentropy(from_logits = TRUE)

model %>% compile(
optimizer = optimizer_rmsprop(learningrate[i]),
loss = loss_fn,
metrics = "accuracy"
)
# when fitting the model we are using the MNIST training data
# and setting epochs to 10 with a validation split of 0.2
fittingmodel <- model %>% fit(x_train, y_train,
epochs = 10, validation_split = 0.2)

# here we are extracting the last of the 10 epochs for us to later
# find the val_accuracy for the model
last_epoch <-length(fittingmodel$metrics$val_accuracy)


val_accuracy <- fittingmodel$metrics$val_accuracy[last_epoch]

# Here the if statement is that we are looking if val_accuracy for the ith model
# is better or worse than the validation accuracy for the model up to the ith model.
# and if we have that the validation accuracy for the ith model is better than we are updating the
# best_val_accuracy to validation accuracy of the ith model.
# else we have that no updating of the best validation accuracy, best model or best hyperparameters.
if (val_accuracy > best_val_accuracy) {
  best_val_accuracy <- val_accuracy
  best_model <- i
  best_hyperparameters <- list(hidden_units = hidden_units[i],
                               dropout = dropout[i],
                                  learningrate = learningrate[i] )
  }
  }

  listan <- list(best_model = best_model, best_hyperparameters,
                 best_val_accuracy = best_val_accuracy)
  return(listan)
}
random_search(hidden_units = hidden_units, dropout = dropout,
              learningrate = learningrate)

$best_model
[1] 4

[[2]]$hidden_units
[1] 32

[[2]]$dropout
[1] 0.04983816

[[2]]$learningrate
[1] 0.001

[[3]]$best_val_accuracy
[1] 0.9875833
```

Based on the output, the hyperparameter settings for the network that yield the best accuracy was Model 4 which number of neurons, dropout rate as well as learning rate set to 32, 0.04983816 and 0.001, respectively. This hyperparameter settings resulted in a accuracy of 0.9875833.

1.6

Now as a next step we will implement a convolutional neural network for the CIFAT10 dataset. Worth mentioning, this network is more complex than the CNN that was implemented for the MNIST dataset above.

```
cifar10 <- dataset_cifar10()
x_train_cif <- cifar10$train$x <- cifar10$train$x/255
x_test_cif <- cifar10$test$x <- cifar10$test$x/255
y_train_cif<- cifar10$train$y
y_test_cif <- cifar10$test$y

model2 <- keras_model_sequential() %>%

  layer_conv_2d(filters = 32, kernel_size = c(3,3),
                padding = "valid", input_shape =  c(32, 32, 3)) %>%
  layer_max_pooling_2d(pool_size = c(2,2)) %>%

  layer_conv_2d(filters = 64, kernel_size = c(3,3), padding = "valid") %>%
  layer_max_pooling_2d(pool_size = c(2,2)) %>%

  layer_conv_2d(filters = 64, kernel_size = c(3,3)) %>%
  layer_flatten() %>%
  layer_dense(units = 64) %>%
  layer_dense(units = 10)

model2
```

```
## Model: "sequential_1"
## ________________________________________________________________________________
##  Layer (type)                       Output Shape                     Param #
## ================================================================================
##  conv2d_4 (Conv2D)                  (None, 30, 30, 32)               896
##  max_pooling2d_2 (MaxPooling2D)     (None, 15, 15, 32)               0
##  conv2d_3 (Conv2D)                  (None, 13, 13, 64)               18496
##  max_pooling2d_1 (MaxPooling2D)     (None, 6, 6, 64)                 0
##  conv2d_2 (Conv2D)                  (None, 4, 4, 64)                 36928
##  flatten_1 (Flatten)                (None, 1024)                     0
##  dense_3 (Dense)                    (None, 64)                       65600
##  dense_2 (Dense)                    (None, 10)                       650
## ================================================================================
## Total params: 122570 (478.79 KB)
## Trainable params: 122570 (478.79 KB)
## Non-trainable params: 0 (0.00 Byte)
## ________________________________________________________________________________
```

Now when the network has been implemented we will fit the model on the CIFAR10 training dataset using 10 epochs and validation split set to 0.2.

```
loss_fn <- loss_sparse_categorical_crossentropy(from_logits = TRUE)

model2 %>% compile(
  optimizer = "adam",
  loss = loss_fn,
  metrics = "accuracy"
)

model2 %>% fit(
  x_train_cif, y_train_cif,
  epochs = 10, validation_split = 0.2
)

1250/1250 [==============================] - 53s 42ms/step - loss: 0.8659 - accuracy: 0.7017 -
  val_loss: 1.1569 - val_accuracy: 0.6193
```

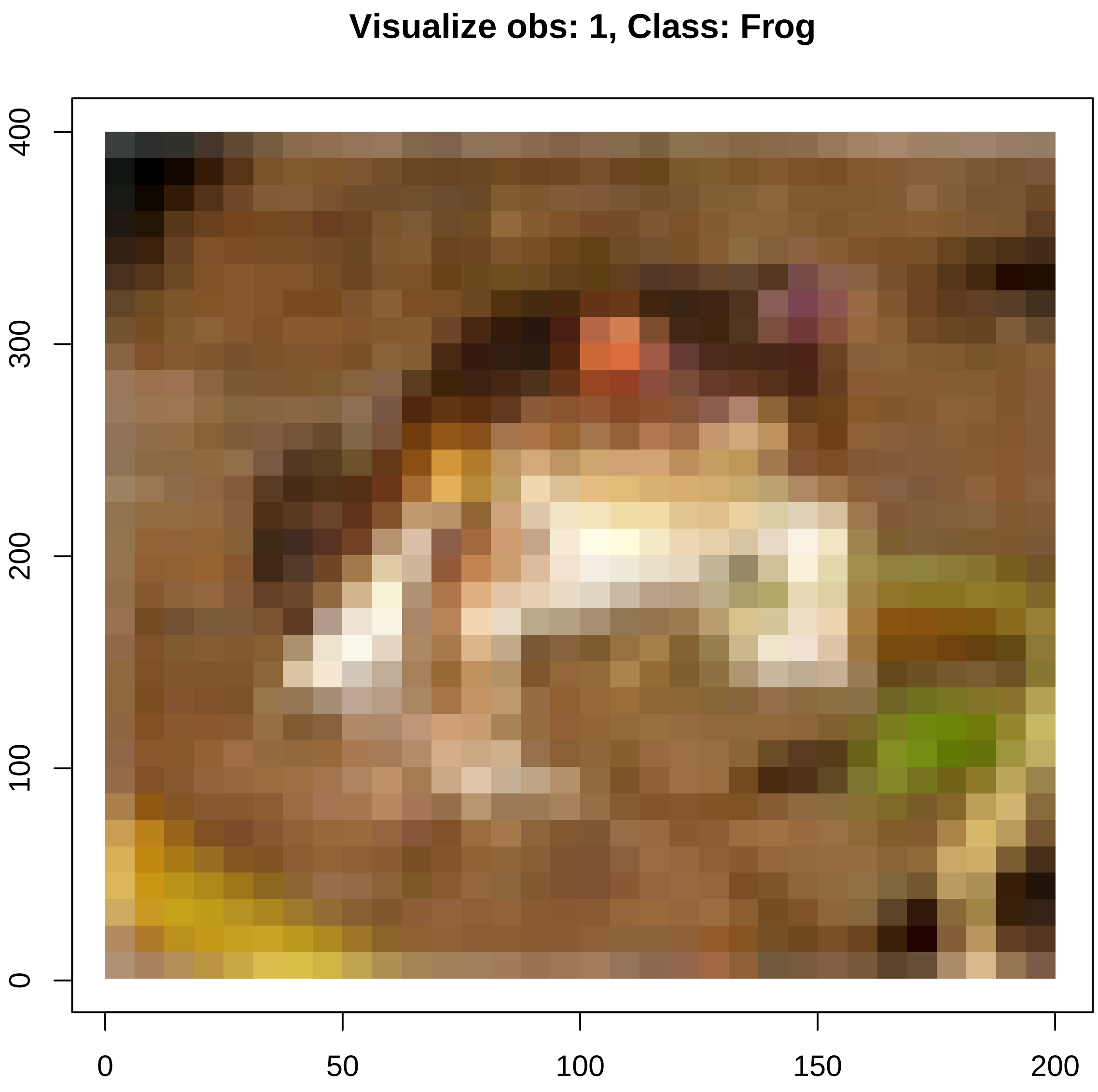
When looking at the accuracy for model2 and comparing it with accuracy for the implemented network on the mnist dataset in section 1.4 then we can see that accuracy for model2 is lower. This is, most likely, due to the increased complexity.

1.7

For this subsection we will visualize one image from the CIFAT10 dataset together its class.

```
par(mfrow = c(1,1))
op <- par(bg = "white")
plot(c(1, 200), c(1, 400), type = "n", xlab = "", ylab = "")
```

```r
rasterImage(x_train_cif[1,,,], xleft = 0, ybottom =  1,
            xright = 200, ytop = 400, interpolate = FALSE)
title("Visualize obs: 1, Class: Frog")
```



1.8

The dataset consists of color images i.e data with a 3 dimensional grid structures. In our case, we have that the first convolutional layer consist of 896 parameters since the total number of parameters for a given convolutional layer can be calculated as follows:

$T = ((3 \times 3 \times 3) + 1) \times 32 = 896$

where T denotes the total number of parameters for a given convolutional layer.

1.9

Now we will try out and compare three different network for the CIFAR10 dataset. The three different network that is implemented below will be named model3, model4, model5.

1.9: Model 3 - adding dropout rate and extra hidden units

For model 3 we have added an dropout rate of 0.4 for the first and the second convolutational layer of the network. Further, we have increased the number of hidden units in the first convolutational layer from 32 to 64.

```r
# model 3
# for this model we increase the number of hidden units for the first convolutional layer to 64.
# Further, we add an dropout rate for the first convolutional layer as well as a max pooling for the
# second convolutional layer.

model3 <- keras_model_sequential() %>%

  layer_conv_2d(filters = 64, kernel_size = c(3,3),
                padding = "valid", input_shape =  c(32, 32, 3)) %>%
  layer_max_pooling_2d(pool_size = c(2,2)) %>%
  layer_dropout(0.4) %>%

  layer_conv_2d(filters = 64, kernel_size = c(3,3), padding = "valid") %>%
  layer_max_pooling_2d(pool_size = c(2,2)) %>%
  layer_dropout(0.4) %>%

  layer_conv_2d(filters = 64, kernel_size = c(3,3)) %>%

  layer_flatten() %>%
  layer_dense(units = 64) %>%
  layer_dense(units = 10)

summary(model3)
```

```
## Model: "sequential_2"
## ________________________________________________________________________________
##  Layer (type)                       Output Shape                      Param #
## ================================================================================
##  conv2d_7 (Conv2D)                  (None, 30, 30, 64)                1792
##  max_pooling2d_4 (MaxPooling2D)     (None, 15, 15, 64)                0
##  dropout_1 (Dropout)                (None, 15, 15, 64)                0
##  conv2d_6 (Conv2D)                  (None, 13, 13, 64)                36928
##  max_pooling2d_3 (MaxPooling2D)     (None, 6, 6, 64)                  0
##  dropout (Dropout)                  (None, 6, 6, 64)                  0
##  conv2d_5 (Conv2D)                  (None, 4, 4, 64)                  36928
##  flatten_2 (Flatten)                (None, 1024)                      0
##  dense_5 (Dense)                    (None, 64)                        65600
##  dense_4 (Dense)                    (None, 10)                        650
## ================================================================================
## Total params: 141898 (554.29 KB)
## Trainable params: 141898 (554.29 KB)
## Non-trainable params: 0 (0.00 Byte)
## ________________________________________________________________________________
```

Now we will fit model3 using the fit function

```
loss_fn <- loss_sparse_categorical_crossentropy(from_logits = TRUE)

model3 %>% compile(
  optimizer = "adam",
  loss = loss_fn,
  metrics = "accuracy"
)

model3 %>% fit(
  cifar10$train$x, cifar10$train$y,
  epochs = 10, validation_split = 0.2
)

1250/1250 [==============================] - 106s 85ms/step - loss: 1.2321 - accuracy: 0.5739 -
  val_loss: 1.2481 - val_accuracy: 0.5716
```

1.9: Model 4 - chaning padding and the kernel size

For model4 we have changed the kernel size from being 3x3 to 4x4 as well as changing padding from being valid to being the same. When we set the padding argument to being valid in the layer-conv-2d function then the dimensions is being reduced but the dimensions stays the same when setting the padding argument to same.

```
# Model 4
model4 <- keras_model_sequential() %>%

  layer_conv_2d(filters = 64, kernel_size = c(4,4),
              padding = "same", input_shape =  c(32, 32, 3)) %>%
  layer_max_pooling_2d(pool_size = c(2,2)) %>%
  layer_dropout(0.4) %>%

  layer_conv_2d(filters = 64, kernel_size = c(4,4), padding = "same") %>%
  layer_max_pooling_2d(pool_size = c(2,2)) %>%
  layer_dropout(0.4) %>%

  layer_conv_2d(filters = 64, kernel_size = c(4,4)) %>%

  layer_flatten() %>%
  layer_dense(units = 64) %>%
  layer_dense(units = 10)

summary(model4)

## Model: "sequential_3"
## ________________________________________________________________________________
##  Layer (type)                    Output Shape                    Param #
## ================================================================================
##  conv2d_10 (Conv2D)                 (None, 32, 32, 64)              3136
##  max_pooling2d_6 (MaxPooling2D)     (None, 16, 16, 64)              0
##  dropout_3 (Dropout)                (None, 16, 16, 64)              0
##  conv2d_9 (Conv2D)                  (None, 16, 16, 64)              65600
##  max_pooling2d_5 (MaxPooling2D)     (None, 8, 8, 64)                0
##  dropout_2 (Dropout)                (None, 8, 8, 64)                0
##  conv2d_8 (Conv2D)                  (None, 5, 5, 64)                65600
##  flatten_3 (Flatten)                (None, 1600)                    0
##  dense_7 (Dense)                    (None, 64)                      102464
##  dense_6 (Dense)                    (None, 10)                      650
## ================================================================================
## Total params: 237450 (927.54 KB)
## Trainable params: 237450 (927.54 KB)
## Non-trainable params: 0 (0.00 Byte)
## ________________________________________________________________________________
```

Now when model4 has been implemented we will fit the model using the fit function

```
loss_fn <- loss_sparse_categorical_crossentropy(from_logits = TRUE)

model4 %>% compile(
  optimizer = "adam",
  loss = loss_fn,
  metrics = "accuracy"
)

model4 %>% fit(
  cifar10$train$x, cifar10$train$y,
  epochs = 10, validation_split = 0.2
)

1250/1250 [==============================] - 157s 126ms/step - loss: 1.1933 - accuracy: 0.5840 -
  val_loss: 1.1211 - val_accuracy: 0.6178
```

1.9: Model 5 - adding one con. layer, max pooling

The main changes we do for model5, compared to the earlier models, is to add one extra convolutional layer as well as max pooling layer for model 5. Further, we assign the relu activation function to the first second and third convolutional layer and softmax activation function to the output layer. Also, we changed some hidden units for the convolutional layers and keeping padding set to valid.

```
model5 <- keras_model_sequential() %>%

  layer_conv_2d(filters = 64, kernel_size = c(4,4),
                padding = "same", input_shape =  c(32, 32, 3), activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2,2)) %>%

  layer_conv_2d(filters = 50, kernel_size = c(4,4), padding = "same", activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2,2)) %>%

  layer_conv_2d(filters = 64, kernel_size = c(4,4), padding = "same", activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2,2)) %>%

  layer_conv_2d(filters = 30, kernel_size = c(4,4)) %>%

  layer_flatten() %>%
  layer_dense(units = 64) %>%
  layer_dense(units = 10, activation = "softmax")

summary(model5)
```

```
## Model: "sequential_4"
## ________________________________________________________________________________
##  Layer (type)                       Output Shape                    Param #
## ================================================================================
##  conv2d_14 (Conv2D)                 (None, 32, 32, 64)              3136
##  max_pooling2d_9 (MaxPooling2D)     (None, 16, 16, 64)              0
##  conv2d_13 (Conv2D)                 (None, 16, 16, 50)              51250
##  max_pooling2d_8 (MaxPooling2D)     (None, 8, 8, 50)                0
##  conv2d_12 (Conv2D)                 (None, 8, 8, 64)                51264
##  max_pooling2d_7 (MaxPooling2D)     (None, 4, 4, 64)                0
##  conv2d_11 (Conv2D)                 (None, 1, 1, 30)                30750
##  flatten_4 (Flatten)                (None, 30)                      0
##  dense_9 (Dense)                    (None, 64)                      1984
##  dense_8 (Dense)                    (None, 10)                      650
## ================================================================================
## Total params: 139034 (543.10 KB)
## Trainable params: 139034 (543.10 KB)
## Non-trainable params: 0 (0.00 Byte)
## ________________________________________________________________________________
```

Now when model5 has been implemented we will fit the model by using the fit function.

```
loss_fn <- loss_sparse_categorical_crossentropy(from_logits = TRUE)

model5 %>% compile(
  optimizer = "adam",
  loss = loss_fn,
  metrics = "accuracy"
)

model5 %>% fit(
  cifar10$train$x, cifar10$train$y,
  epochs = 10, validation_split = 0.2
)
1250/1250 [==============================] - 139s 111ms/step - loss: 0.7515 - accuracy: 0.7365 -
  val_loss: 0.9294 - val_accuracy: 0.6827
```

1.9: Compare the different models

When looking at the loss and the accuracy, as well as their validation estimates, we can see that model5 performed best on the training dataset.

For model 3 we added a dropout rate of 0.4 to the first and second convolutional layer and extra hidden units for the first convolutional layer. We did so because by adding extra hidden units we increase the representational capacity of the model and we included dropout to aviod units to conspire thus overfitting the training data.

For model4 we changed the kernel size from 3x3 to 4x4 and changed to padding in the network. We increased the kernel width because by doing so we reduce capacity of the model and reducing the memory cost. Padding was included to keep the representation size large at each layer of the network.

For model5 we added one extra convolutional layer and a max pooling layer. We did so since by adding extra layers into the model, by adding layers into the model we increase its representational capacity where the model is more capable of representing more complicated functions and problems.

Task 2: Implementing a convolutional layer

For this task we will implement a layer of a convolutional neural network with one filter without training it but only implementing. We start by importing examples from the MNIST dataset as follows:

```r
library(uuml)
data("mnist_example")
```

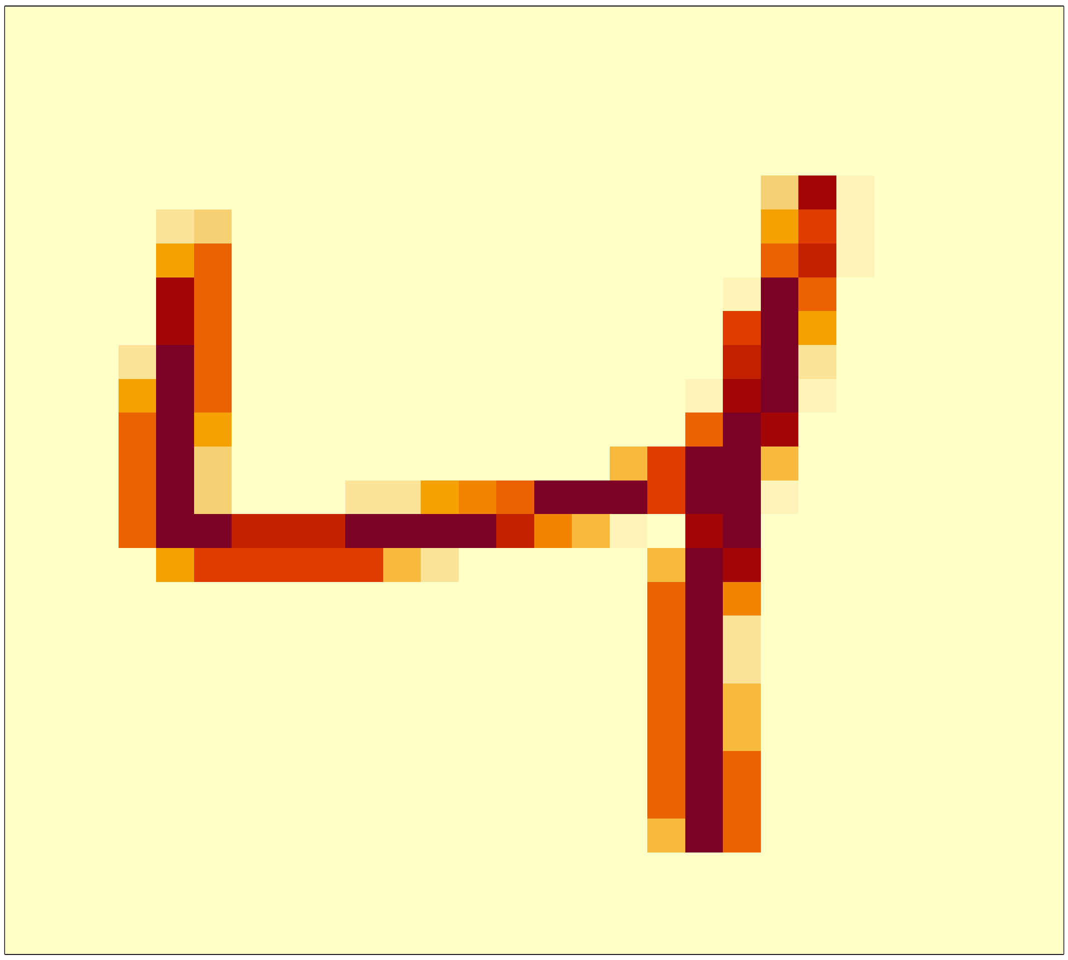
and we have been given a chunk of code to visualize an image, the code can be found in the instructions for Assignment 4 in the Machine learning course.

2.1

For subtask 2.1, we will visualize digit 4 in the MNIST example.

```r
par(mfrow = c(1,1))
im <- mnist_example[["4"]]
image(1:ncol(im), 1:nrow(im), im,
xlab = "", ylab = "",
xaxt='n', yaxt='n', main="Digit 4 from MNIST example")
```

Digit 4 from MNIST example



2.2

For subtask 2.2 we will now implement a convolution function that takes an input as an MNIST image, denoted as X, and an arbitrary large square kernel, denoted as K. The convolution function will then return a valid feature map.

Before we implement the convolution function we need to state the dimensions of the feature map which is a function of the dimensions of X and the dimensions of K. Let K: $d \times d$, X:$n \times p$ and Y be the squared kernel, the MNIST image input and the feature map, respectively. Thus, the dimensions of the feature map will correspond to:

$m = n - d + 1$
$c = p - d + 1$

where $Y : m \times c$. For the case when we have $X : 4 \times 4$ and $K : 3 \times 3$, we calculate m and c to be:

$m = 4 - 3 + 1 = 2$
$c = 4 - 3 + 1 = 2$

Note that only the kernel matrix needs to be squared while that is not the case for the MNIST image input X. However, Y will not necessary be squared as well.

```
# dimensions: 4x4
X <- mnist_example[["4"]][12:15,12:15]
# dimensions 3x3
K <- matrix(c(-0.5, -0.5, -0.5, 1, 1, 1,
              -0.5, -0.5, -0.5), nrow = 3, byrow = TRUE)

convolution <- function(X, K){
  # given the dimensions for X and K we have that the
  # dimensions for the feature map corresponds to 2x2

  Y_col <- ncol(X) - ncol(K) + 1
  Y_row <- nrow(X) - nrow(K) + 1
  Output <- matrix(NA, ncol = Y_col, nrow = Y_row)

  for ( i in 1:Y_row) {
  for ( j in 1:Y_col) {

    # Here we are doing the indexing on X with respect to
    # the dimensions for K and creating some type of
    # block matrix.
    index.row <- seq(from = i, to = i + nrow(K) -1, by = 1)
    index.col <- seq(from = j, to = j + ncol(K) -1, by = 1)

    # Here we index X creating a subset of the total matrix
    # and taking as.vector on both X and K to easily be able to
    # multiply the two objects element-wise and sum them together
    X_k <- as.vector( X[index.row, index.col] )
    K_k <- as.vector(K)
    # lastly we taking the sum for each element in the output /
    # feature map object.
    Output[i,j] <- sum(X_k * K_k)
    }
  }
  return(Output)
}
convolution(X,K)
```

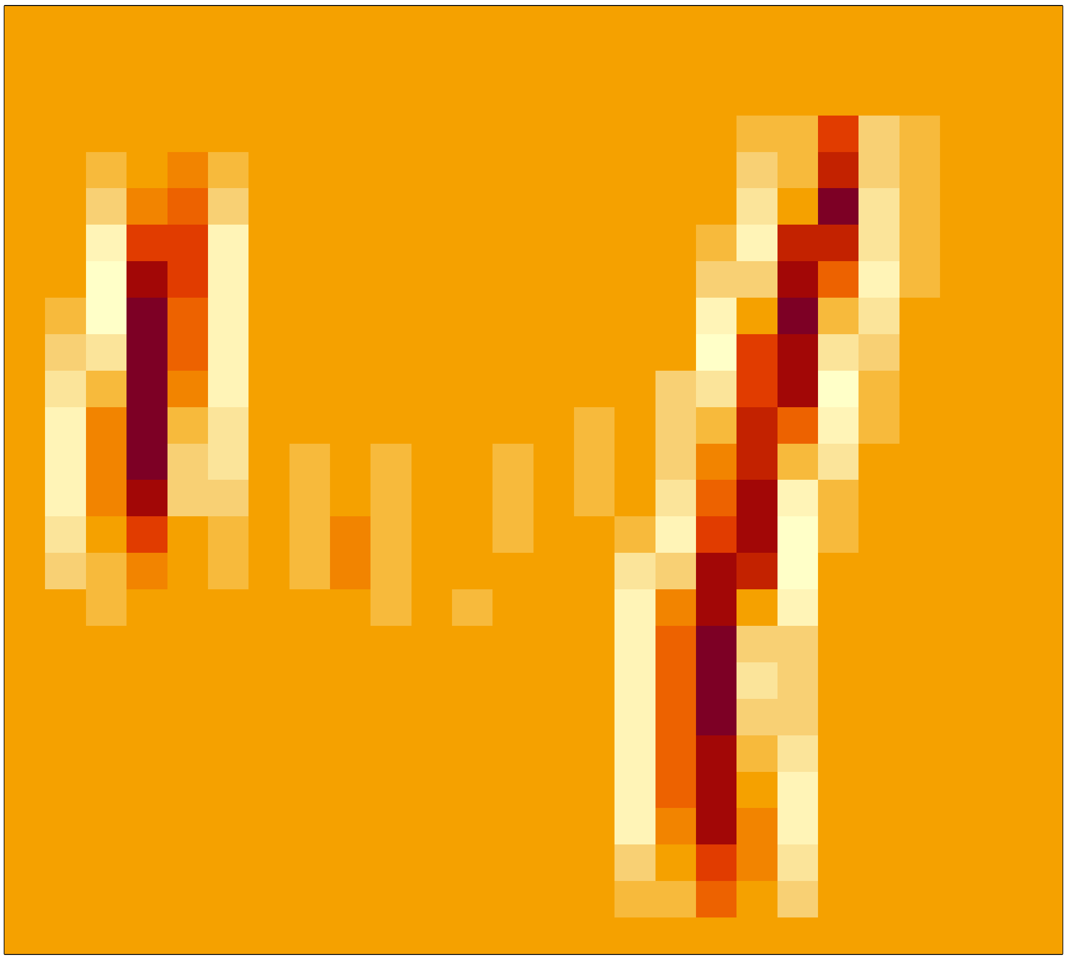
```
##      [,1] [,2]
## [1,]   -1   27
## [2,]  -36  -36
```

2.3

Now when the convolution function has been implemented we will visualize the feature map of digit 4 from MNIST example using the above kernel K. To visualize this feature map we will use the code chunk that was given in the begin of this task.

```r
X1 <-  mnist_example[["4"]]
im <- convolution(X1, K)
image(1:ncol(im), 1:nrow(im), im,
xlab = "", ylab = "",
xaxt='n', yaxt='n', main="Feature map of digit 4")
```

Feature map of digit 4



2.4

Now we implement all steps in a convolutional-layer function that takes the kernel, bias activation function. In mathematical notations, this can be expressed as $\mathbf{h} = \sigma(\mathbf{XK} + b)$ where X, K, b, σ denotes the input image, the kernel matrix, the bias and the activation function, respectively. The function below uses relu as the activation function.

```
relu <- function(x) max(0, x)
```

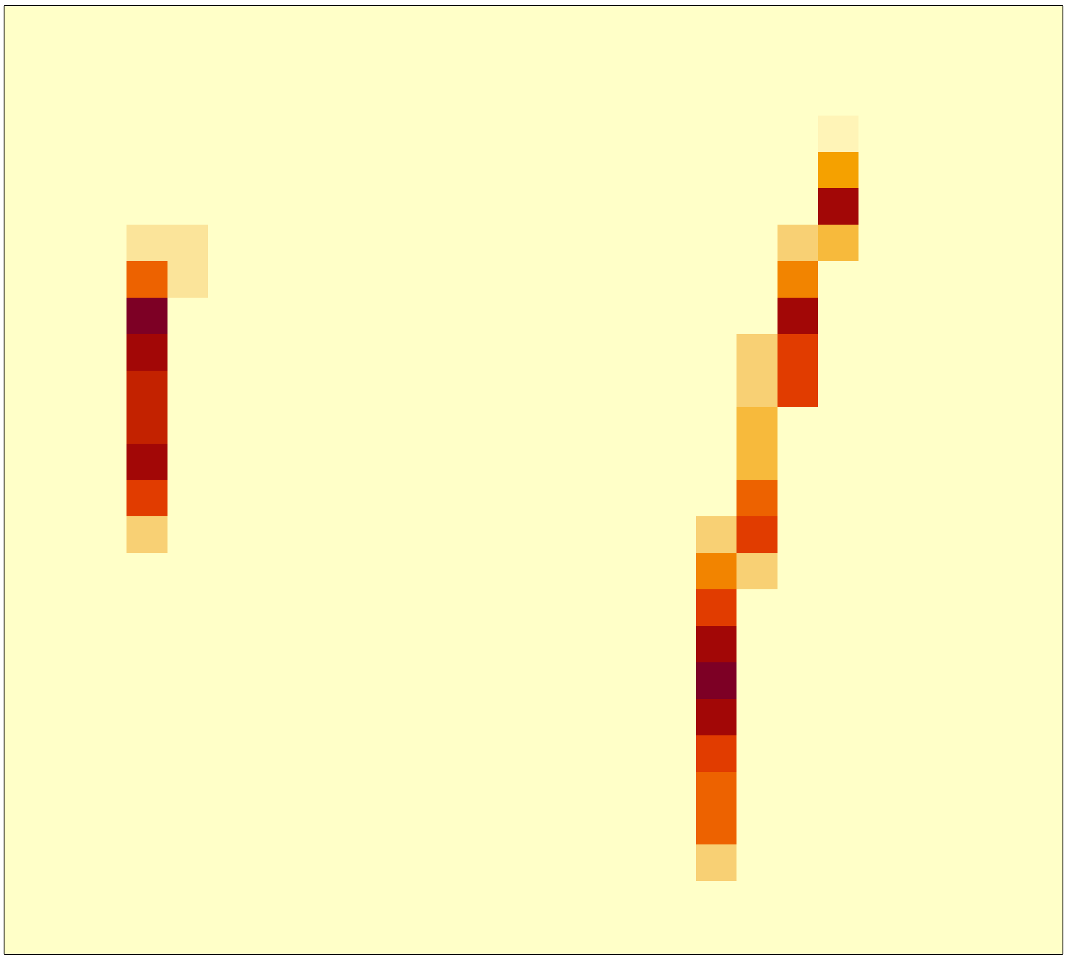
```
convolutional_layer <- function(X, K, b = 100, relu) {

  Output <- apply(convolution(X,K) + b, MARGIN = c(1,2), relu)
  return(Output)
}
convolutional_layer(X, K, b = 100, relu)
```

```
##      [,1] [,2]
## [1,]   99  127
## [2,]   64   64
```

2.5

```
X1 <-  mnist_example[["4"]]
im <- convolutional_layer(X1, K, b = -150, relu)
image(1:ncol(im), 1:nrow(im), im,
xlab = "", ylab = "",
xaxt='n', yaxt='n', main="Digit 4 with org kernel matrix")
```

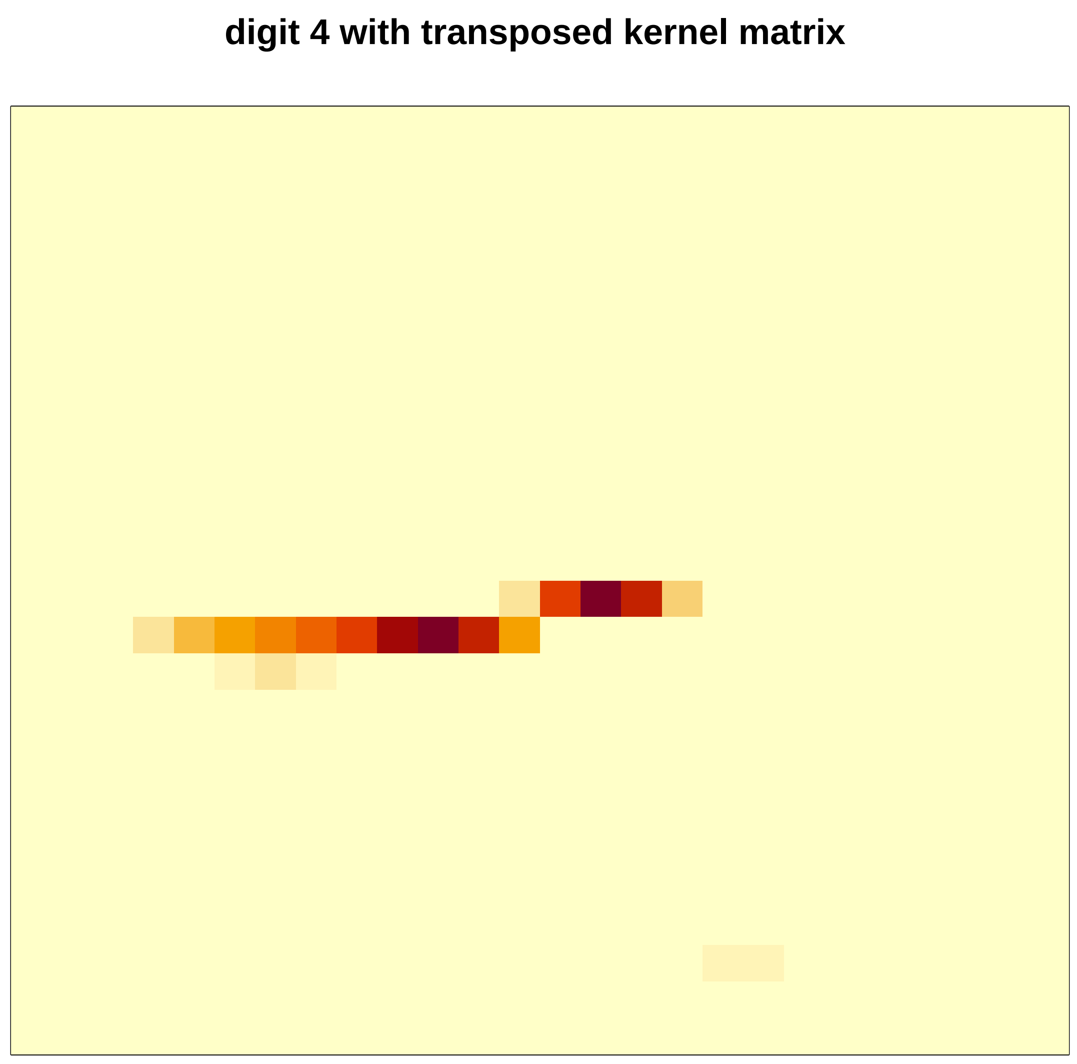
Digit 4 with org kernel matrix



Depending on the structure of the Kernel filter different type of edges will be captured and in our case we have that the feature map seem to capture the vertical lines of the handwritten digit 4 from MNIST example.

2.6

```r
X1 <-  mnist_example[["4"]]
im <- convolutional_layer(X1, t(K), b = -150, relu)
image(1:ncol(im), 1:nrow(im), im,
xlab = "", ylab = "",
xaxt='n', yaxt='n', main="digit 4 with transposed kernel matrix ")
```



When taking the transpose of the kernel filter, it seems that the feature map seem to capture the horizontal line of the handwritten digit 4 from MNIST example.

2.7

As a last step to the convolutional layer function we will implement a 2×2 stride max pooling layer. The function takes on a MNIST image and creating 2×2 block matrices and extracting the element with the max value within the block matrix.

```
X <- mnist_example[["4"]][12:15,12:15]

maxpool_layer <- function(X) {

  pool_col <- ncol(X)/2
  pool_row <- nrow(X)/2

  Output <- matrix(NA, ncol = pool_col, nrow = pool_row)

  for ( i in seq(from = 1, to = nrow(X), by = 2)) {

    for ( j in seq(from = 1, to = ncol(X), by = 2)) {

      indexing_col <- seq(from = 1, to = ncol(X), by = 0.5) [j]
      indexing_row <- seq(from = 1, to = ncol(X), by = 0.5) [i]
      Output[indexing_row,indexing_col] <- max(X[i:c(i+1), j:c(j+1)])
    }
  }
  return(Output)
}
maxpool_layer(X)
```

```
##      [,1] [,2]
## [1,]  250  144
## [2,]  198  241
```

2.8

Now we will put all the implemented functions together and visualize the final output of the own convolutional layer

```
X1 <- mnist_example[["4"]]
relu <- function(x) max(0, x)
im <- maxpool_layer(convolutional_layer(X1, K, -370, relu))
image(1:ncol(im), 1:nrow(im), im,
xlab = "", ylab = "",
xaxt='n', yaxt='n', main="Digit 4 when including max pooling")
```

Digit 4 when including max pooling