Facial Keypoint Detection - Documentation

Henry Perschk, Lucas Stegger

7 1 2017

Contents

1 Introduction

The fkdR (short for Facial Keypoints Detection in R) package is intended for predicting facial keypoint positions like the centers of the eyes or the nose tip for example. It furthermore provides functionality to train models for facial keypoint recognition. This is achieved by using the mlR package to make use of a standardized interface for e.g. resampling or tuning. As of now we implemented a patch search learner with mlR to show the concept of our package and the way image data is handled. Our research showed, that best results for image recognition by machine-learning are achieved by Deep Neural Nets. Thus, we moreover composed first a multi-layer-perceptron as well as a convolutional neural net, both based on tensorflow. The data we use in this package is based on the Facial Keypoints Detection contest posted on Kaggle.

- structure of package?
- structure of documentation

2 Loading and Preprocessing the Data

- structure of data
- 96 x 96 pixel
- as column-wise vector
- NAs
- histogram equalization
- outlook
- transformation in 1-dimensional data

3 Patch Search Learner

The patch search learner we implemented is based on the Getting Started Tutorial from the Facial Keypoints Detection contest. We used the logic and realized it using `mlR`, which allowed us to make use of `mlR`'s standardized interface for machine learning. Thus, we tried to improve the results by making use of tuning and resampling. As usual the patch search algorithm can be devided into training and prediction.

The basic idea behind the patch search learner is to extract all pixels with a specific radius around each keypoint in each image. The extracted pixels are refered to as patch. Afterwards, for each keypoint the mean patch is calculated by calculating the mean of each pixel. This mean patch can in a next step be used on any image containing the keypoint (assuming it is also a 96x96 pixel grayscale image, where the face ideally covers the whole width and height of the image and is horizontally aligned). Predicting the keypoint is achieved by running through the desired image pixel by pixel and again extracting the the patch around this pixel. The point, where the extracted patch best matches the trained mean patch is used as prediction.

3.1 Training a Patch for a Facial Keypoint

As indicated the first step is to extract the patch for each keypoint from each training image. One limitation using mlR was, that mlR as of now does not offer an interface for regression with more than one targets. This becomes a problem, as we are trying to train and predict 15 keypoints which consist of two coordinates. As explained in section 2, we solved the problem of predicting two coordinates by transforming the two coordinates into one value. As long as the width and height of the images is known, this value can easily be converted back to two coordinates.

The problem of having 15 different keypoints can be solved by training 15 models for only one keypoint each. Normally, this can be a limitation, because the information where the eyes lie for example might give us some information about the location of the nose tip. As the Patch Search Learner is not able to respect these interdependencies, we can ignore this disadvantage. However, it has to be kept in mind and one might have to think of a different solution when using advaned learners, that are able to respect these interdependencies. One solution could be to extend the mlR functionality by an interface for multi-target regression.

Furthermore, all patches are squares where the center point is the keypoint and the width as well as height are calculated by `patch_size * 2 + 1`. This allows easier handling and storing of the patch as it would be the case with a circle. The variable `patch_size` is therefore the first of two hyperparameters the Patch Search Learner provides.

Based on the aforementioned assumptions the learner is implemented. First, a new mlR learner is initialized by ‘makeRLearnerRegr()‘as a regression learner. As indicated in the code below the two hyperparameters are defined in this function. The ’search_size’ hyperparameter is relevant for the predictions and will therefore further explained during the section 3.2.

```
makeRLearner.regr.patchSearch = function() {
  makeRLearnerRegr(
    cl = "regr.patchSearch",
```

```
    package = "fkdR",
    par.set = makeParamSet(
      makeNumericLearnerParam(id = "patch_size", default = 10, lower = 0, tunable = TRUE),
      makeNumericLearnerParam(id = "search_size", default = 2, lower = 1, tunable = TRUE)
    ),
    properties = c("numerics"),
    name = "Patch Search Learner for Keypoint Detection in Images",
    short.name = "patchSearch",
    note = ""
  )
}
```

Next step is to define the regression part of the learner. To make it visible for `mlR` the function name has to be constructed of the term "trainLearner" and the name we defined in the aforementioned `makeRLearnerRegr()` function. Relevant data are passed to our custom function for trinaing the mean patch for the current keypoint. These are the formula of the current task (containing the name of the target keypoint) and the training data.

```
trainLearner.regr.patchSearch = function(.learner, .task, .subset, ...) {
  patchSearch.train(f = getTaskFormula(.task),
                    d.tr = getTaskData(.task, .subset),
                    ...)
}
```

Respecting the assumptions from the beginning of this section training data has to be a data frame with two columns, the first containing the images as vector, the second containing the desired keypoint as single value. Next to these training data the task formula and hyperparameters are parameters of the custom Patch Search training function. The desired keypoint is extracted from the formula by using `all.vars(f)[1]`.

```
patchSearch.train <- function(f, d.tr, patch_size = 10, search_size = 2) {
  coord = all.vars(f)[1]
  cat(sprintf("computing mean patch for %s\n", coord))
```

Next step is to extract all patches for the current keypoint. The following code runs through all training images and first converts them to a 96x96 matrix. Afterwards the keypoint value is transformed back to x and y coordinate. In a next step the outer coordinates of the patch can be calculated using the `patch_size` hyperparameter. All patches are returned as vector and stored in the variable `patches`. We are using the `doParallel` package to speed up the calculation.

```
  # extract all patches
  patches <- foreach (i = 1:nrow(d.tr), .combine=rbind) %do% {
    if ((i %% 100)==0) { cat(sprintf("Extracting %s patch from training image %d/%d\n",
                                     coord, i, nrow(d.tr))) }
    im  <- matrix(data = d.tr[i,"Image"], nrow=96, ncol=96)

    # transform to x and y coordinate
    xy  <- d.tr[i, coord]
    x   <- xy %/% 96 + 1
    y   <- xy %% 96

    # determine outer coordinates of patch
    x1  <- (x-patch_size)
    x2  <- (x+patch_size)
    y1  <- (y-patch_size)
    y2  <- (y+patch_size)
    if ( (!is.na(x)) && (!is.na(y)) && (x1>=1) && (x2<=96) && (y1>=1) && (y2<=96) )
```

```
    {
      as.vector(im[x1:x2, y1:y2])
    }
    else
    {
      NULL
    }
  }
```

As soon as all patches are extracted, we can calculate the mean of them. At this point, we decided to plot the mean patch to get some insights in the mean patch calculation. Figure 1 shows one of these mean patches. Although it is based on so many different faces, one can still clearly see a mouth and the bottom part of a nose on this figure. By returning the mean patch at the end of the function it is stored as the model internally in `mlR` and can be accessed during the predictions.

```
  # return mean patch
  mean.patch = matrix(data = colMeans(patches), nrow=2*patch_size+1, ncol=2*patch_size+1)

  # plot mean patch
  par(mar = rep(0, 4))
  image(1:(2*patch_size+1), 1:(2*patch_size+1),
        mean.patch[nrow(mean.patch):1,ncol(mean.patch):1],
        col=gray((0:255)/255), xaxt = "n", yaxt = "n", ann = FALSE, breaks = 0:256)

  # return the mean patch
  mean.patch
}
```

3.2 Predicting a Facial Keypoint using a Patch

Having computed the mean patch for a particular keypoint, we can start with predictons. For the predictions we run through the images and search for the best match of the mean patch. As it would be very inefficient and expensive to compare the match to all possible patch positions, the Patch Search Learner only searches around the mean keypoint positions of the training dataset. The size of the area around this mean keypoint, where the learner searches, is determined by the hyperparameter `search_size`. Similar to the identification of the outer coordinates of the patch, the search area is determined by setting its center to the mean position and the width and height to `2 * search_size + 1`. Afterwards the algorithm runs through every pixel in the search area and the mean patch's center is set to the pixel in order to calculate the match. The position in the search area with the best match is used as prediction.

Again we are using the `mlR` interface to provide the functionality. Therefore, we define the following function to make the prediction part of the learner available to `mlR`. Before handing over the data to our custom function, we extract the hyperparameters of the current model and to be sure they have some value set them to the default values in case they are `NULL`.

```
predictLearner.regr.patchSearch = function(.learner, .model, .newdata, ...) {
  .patch_size = .model$learner$par.vals$patch_size
  .search_size = .model$learner$par.vals$search_size

  .patch_size = ifelse(is.null(.patch_size), 10, .patch_size)
  .search_size = ifelse(is.null(.search_size), 2, .search_size)

  patchSearch.predict(.model,
                      f = getTaskFormula(.model$task.desc),
```

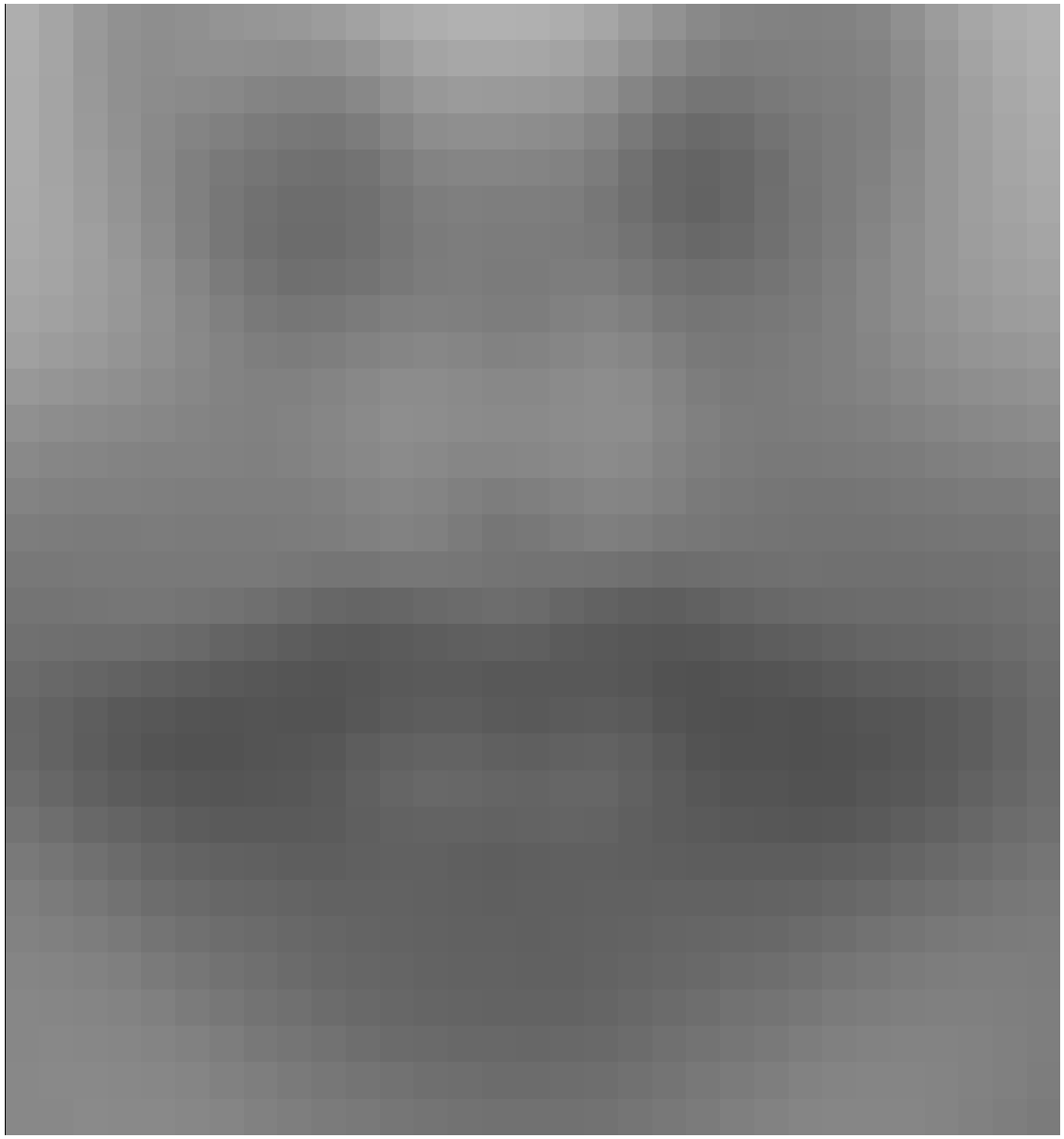

Figure 1: Mean patch for mouth center top lip with ‘patch_size = 15’

```
                      d.te = .newdata,
                      patch_size = .patch_size,
                      search_size = .search_size,
                      ...)
}
```

Our custom function accepts the model, the formula, test data and the hyperparameters as parameters. First, the mean patch is extracted from the model and the target variable name is read from the formula.

```
patchSearch.predict <- function(model, f, d.te, patch_size = 10, search_size = 2)  {
  mean.patch = model$learner.model

  # the coordinates we want to predict
  coord = all.vars(f)[1]
  coord_x <- paste(coord, "x", sep="_")
  coord_y <- paste(coord, "y", sep="_")
```

Now the mean of the x and y coordinate is calculated based on the training data. By using the hyperparameter `search_size` we can calculate the search area. As the mean positions of a keypoint could lie so close to the border, that the patch would not lie completely inside the image, we have to cut the search area by these pixels. The variable `params` then contains all coordinates in the search area.

```
  # the average of them in the training set (our starting point)
  mean_x  <- mean(d.train[, coord_x], na.rm=T)
  mean_y  <- mean(d.train[, coord_y], na.rm=T)

  # search space: 'search_size' pixels centered on the average coordinates
  x1 <- as.integer(mean_x)-search_size
  x2 <- as.integer(mean_x)+search_size
  y1 <- as.integer(mean_y)-search_size
  y2 <- as.integer(mean_y)+search_size

  # ensure we only consider patches completely inside the image
  x1 <- ifelse(x1-patch_size<1,  patch_size+1,  x1)
  y1 <- ifelse(y1-patch_size<1,  patch_size+1,  y1)
  x2 <- ifelse(x2+patch_size>96, 96-patch_size, x2)
  y2 <- ifelse(y2+patch_size>96, 96-patch_size, y2)

  # build a list of all positions to be tested
  params <- expand.grid(x = x1:x2, y = y1:y2)
```

We can now run through all test images and search for the best match of the models mean patch in the search area. The outer loop is responsible for running through the images, whereas the inner loop runs through all coordinate combinations in the search area. For each coordinate combination the patch with the same size as the models mean patch is extracted. Both, the extracted patch and the mean patch are then converted to a vector and the match between them is calculated by the correlation amongst them. Afterwards, the coordinates and the correlation is stored in a data frame, such that we can easily extract the coordinates with the highest correlation value. The final step is to retransform the two coordinates back to a single value to make them compatible with the general data format we agreed upon. Thus, we have our prediction.

```
  # for each image...
  r <- foreach(i = 1:nrow(d.te), .combine=rbind) %do% {
    im <- matrix(data = d.te[i, "Image"], nrow=96, ncol=96)
    if ((i %% 100)==0) { cat(sprintf("Predicting %s for test image %d/%d\n",
                                     coord, i, nrow(d.te))) }
```

```
    # ... compute a score for each position ...
    r  <- foreach(j = 1:nrow(params), .combine=rbind) %do% {
      x     <- params$x[j]
      y     <- params$y[j]
      p     <- im[(x-patch_size):(x+patch_size), (y-patch_size):(y+patch_size)]
      score <- cor(as.vector(p), as.vector(mean.patch))
      score <- ifelse(is.na(score), 0, score)
      data.frame(x, y, score)
    }

    # ... and return the best
    best <- r[which.max(r$score), c("x", "y")]
    best
  }

  # retransform to single value
  result <- (round(r[1]) - 1) * 96 + round(r[2])
  result[1]$x
}
```

4 Tensorflow in Combination With R

Tensorflow (TF) is an open source library for machine learning, originally developed by Google. It became extremely popular recently, with about 42,500 stars and 20,000 forks on Github.

The main principle behind TF are so called *computational data flow graphs*. A machine learning model (like e.g. a neural network) is described in its enirety through such a graph. All nodes in that graph are placeholder objects. Only after the whole graph has been initialized, actual data is fed into the nodes. This makes it easier to quickly change the structure of the model, without having to worry about keeping the model with all weights etc. in memory.

Initially, Google provided an API for Python and C++, but the team behind RStudio recently released a package, that provides a TF Interface for R. Although the project is still in a quite early stage, we decided to work with this package in order to get a flexible library for deep learning projects - which are surprisingly rare in R. We were eager to be part of the very early adopters of the R/Tensorflow combination, so that we even submitted some error fixes on Github in order to get the package to work on our system.

4.1 Deep Neural Nets for Facial Keypoints Detection

Since we want to predict 15 keypoints in the face of a person, each having an x- and a y-coordinate, the prediciton of the statistical model has to provide 30 values. Moreover, the facial keypoints detection problem is a regression problem: the predicted number for a certain keypoint position contains spatial information, which would be lost if the prediction would be treated as a classification problem (i.e. no numerical order of the values). As the analyzed data set's images have the dimensions [96, 96], the predicted values should also lie in that range.

The nature of multi-node layers in neural networks makes it easy to obtain multiple outputs (as compared to other model-types where it might be necessary to create one model for every prediction value). In addition, especially convolutional neural networks have proven to work very well in image related problems (and more generally in problems where spatial information matters). We have built and successfully trained a multi-layer perceptron and a convolutional neural network, which will be described in the following.

4.2 Multi-Layer Perceptron (MLP)

4.3 Data Preparation

After loading the tensorflow package

```
library(tensorflow)
```

the data has to be prepared in order for TF to process it. First, the data has to be in the matrix data format.

```
train.x = as.matrix(d.train$Image)
train.y = as.matrix(d.train[,-31])
```

TF expects input to be scaled to values in the interval [0, 1], so we scale the pixel intensities (i.e. feature values) by dividing by 255 (where 0 is a black pixel and 255 is a white pixel).

```
train.x = train.x / 255
```

On the middle of the image, we place the origin of the coordinate system, such that the left (/down) most point is -1 and the right (/up) most point is +1. Each of the target values is scaled into this [-1, 1] interval (from prior [0, 96]).

```
train.y = (train.y - 48) / 48
```

4.4 Hyperparameters

We control four different hyperparameters, that can be adapted once we decided for a certain network structure:

- Learning Rate. Controls how big of a step the optimization algorithm should take in every training step. High values mean potentially faster results from training, but might miss out on optima, by "overstepping" them.
- Number of training epochs. One epoch marks one run-through of all training data through the training process.
- Batch size. From the training data, a random batch is drawn on every training step. Only this data is then used to further train the network. After one training epoch, all batches in the training set have been used.
- Display Step. How often we want to log the current status.

```
learning_rate = 0.001
training_epochs = 1000L
batch_size = 50L
display_step = 1L
```

4.5 Network Structure

We experimented with different depths of the network but realized quickly that training on a CPU (as opposed to a GPU) puts certain limits on the complexity. Nevertheless, we achieved decent results with the following structure (in combination with the above hyperparameters).

```
n_input = 9216L # 96x96 pixels
n_hidden_1 = 256L # 1st layer number of features
n_hidden_2 = 256L # 2nd layer number of features
n_classes = 30L # 15 x, 15 y coordinates
```

4.6 Tensorflow Graph

First we create some placeholder variables to hold the input (x) and the output (y).

```
x = tf$placeholder(tf$float32, shape(NULL, n_input))
y = tf$placeholder(tf$float32, shape(NULL, n_classes))
```

Then we create two convenience functions to simplify the creation of weight and bias variables. These variables get initialized with a random number, whose extent can be modified via the *stddev* parameter.

```
weight_variable <- function(shape) {
  initial <- tf$truncated_normal(shape, stddev=0.1)
  tf$Variable(initial)
}

bias_variable <- function(shape) {
  initial <- tf$constant(0.1, shape=shape)
  tf$Variable(initial)
}
```

Afterwards we define the actual model in the graph. (Hidden) Layer 1 is a matrix multiplication of the input x. The shape is defined by the parameters specified in the **Network Structure** section. The output of this layer runs through the *ReLU* (*Rectified Linear Unit*) activation function $f(x) = max(0, x)$, which sets all negative output to 0. The output is fed into the second (hidden) layer, which performs another matrix multiplication and has the previosuly defined shape. The activation function is again a ReLU. Finally, the output is fed into the output layer, which has exactly 30 nodes to represent all 15 coordinate pairs of the facial keypoints.

```
layer1 = tf$add(tf$matmul(x, weight_variable(shape(n_input, n_hidden_1))),
                bias_variable(shape(n_hidden_1)))
layer1 = tf$nn$relu(layer1)

layer2 = tf$add(tf$matmul(layer1, weight_variable(shape(n_hidden_1, n_hidden_2))),
                bias_variable(shape(n_hidden_2)))
layer2 = tf$nn$relu(layer2)

out_layer = tf$matmul(layer2, weight_variable(shape(n_hidden_2, n_classes))) +
            bias_variable(shape(n_classes))
```

Now we define the cost function of the optimization method, which is the MSE. For the optimization method itself we used the popular Adam Optimizer, which usually converges faster than many other optimizers such as stochastic gradient decent. The accuracy measure used in the Kaggle competition is the RMSE. Therefore we can get an estimate of our performance on test data by taking the square root of the MSE. We multiply by 48 in order to account for the previous scaling of the training target data.

```
cost = tf$reduce_mean(tf$square(out_layer - y))
optimizer = tf$train$AdamOptimizer(learning_rate = learning_rate)$minimize(cost)
accuracy = tf$sqrt(cost) * 48
```

Finally, this graph definition can not be interfaced until a TF session is launched and the TF variables we just created are initialized.

```
sess <- tf$Session()
sess$run(tf$initialize_all_variables())
```

4.7 Training and Evaluating the Model

First we define a function that makes it easy to pass batches into the training mechanism.

```
nextBatchIndices <- function(indices, batchNr, batch_size) {
  position = batchNr * batch_size - batch_size + 1
  if ((position + batch_size) > length(indices)) {
    return(indices[position:length(indices)])
  }
  return(indices[position:(position + batch_size - 1)])
}
```

The number of batches for the training set is indirectly given by the hyperparameter *batch_size*.

```
numberOfBatches = ceiling(nrow(train.x) / batch_size)
```

The training process takes place in two nested loops. The outer loop simply cycles for the specified number of training epochs, every time shuffling the indices of the training set in order to provide "fresh" batches for training, which helps regularizing (reduce overfitting) the network. In the inner loop the previosuly defined function is used to get the next batch of data, which is then used for training. In addition the current status of the progress is printed to the terminal.

```
for(epoch in seq_len(training_epochs)) {
  shuffledIndices = sample(seq_len(nrow(train.x)))

  for(batchNr in seq_len(numberOfBatches)) {
    rowIndices = nextBatchIndices(shuffledIndices, batchNr, batch_size)

    train_accuracy <- sess$run(accuracy, feed_dict = dict(x = train.x[rowIndices, ], y = train.y[rowIndi
    cat(sprintf("Epoch: %d | Batch: %d/%d | Training RMSE: %g\n", epoch, batchNr, numberOfBatches, trai

    sess$run(optimizer, feed_dict = dict(x = train.x[rowIndices, ], y = train.y[rowIndices, ]))
  }
}
```

After training is completed, one could assess the estimated performance on test data. This is not really necessary, as Kaggle provides us with a concrete accuracy score. However, using suffienctly large batch sizes can yield reasonable performance estimations just using the current batch already during training (like we do in the code above). Due to the shuffling + use of multiple epochs (emulates *replacement*) this is very similar to the bootstrap approach. For quick evaluations, one could also just split the training set into *train* and *test* and then feed the test data into the TF model.

```
test_accuracy <- sess$run(accuracy, feed_dict = dict(x = test.x, y = test.y))
cat(sprintf("Test RMSE: %g", test_accuracy))
```

This test data can then be used to actually plot the output of the neural net on one of the corresponding images. The multiplications serve the purpose of rescaling.

```
data = test.x * 255
pred = sess$run(out_layer, feed_dict = dict(x = test.x)) * 48 + 48
fkdR::plotFacialKeypoints(data, 1, pred)
```

4.8 Submission File

Using the fkdR package, creating a submission file for the Kaggle competition only takes three simple steps. First, the test data set has to be scaled in order to be processed by our TF model. Second, the model has

to predict the target value by feeding it the test data and scaling it back to the original scale. Third, the *writeSubmissionFile* function has to be called.

```
data = d.test$Image / 255
pred = sess$run(out_layer, feed_dict = dict(x = data)) * 48 + 48
fkdR::writeSubmissionFile(predictions = pred, "/path/for/submissions/")
```

4.9 Save and Restore

Finally, we made an effort to save and restore the TF model (including the trained weights and biases), so that there is no data loss after restarting the R session.

```
# Save data
saver <- tf$train$Saver()
data_file <- saver$save(sess, paste0("/path/for/submissions/", "fkdr_mlp_1000epochs.ckpt"))

# Restore Data
sess = tf$Session()
restorer = tf$train$import_meta_graph(paste0("/path/for/submissions/", "fkdr_mlp_1000epochs.ckpt.meta"))
restorer$restore(sess, tf$train$latest_checkpoint("/path/for/submissions/"))
```

Unfortunately, this (as of now) leads to unexpected behavior, although the code should be correct. We filed a Github issue for this and are in a conversation with two of the lead RStudio/Tensorflow programmers.

5 Convolutional Neural Network (CNN)

In order to build a convolutional neural network, we need two more convenience functions and of course a different layer structure.

5.1 Convolution, Max Pooling, Dropout and Fully Connected Layer

In the following, the process of convolution and pooling will be described in general, followed by the code that implements the theoretical excursion.

5.1.1 Theory

5.1.1.1 Convolution

A convolutional layer is essentially a set of so called filters, which can be trained. In contrast to traditional image recognition techniques, where filters like the *Sobel Edge Detector* might be used to detect certain features in an image, the filters in the convolutional layer are randomly initialized (not predefined) and they are solely shaped through training the network.

A filter is essentially a window (typically 5 by 5 pixels) that slides (–> **convolves**) across an image. For colored images, the window is a cube (5x5x**3**), since the image contains three color channels.

Once a network is trained sufficiently, one of the filters might actually look like an edge (similar to Sobel for instance). The following picture shows an exemplary set of learned filters that can detect edges and other important features in images.

When using the trained network to make a prediction, the filter window slides across the image. Every pixel in the filter window gets matched with the underlying part of the image. If the pixels are similar, this

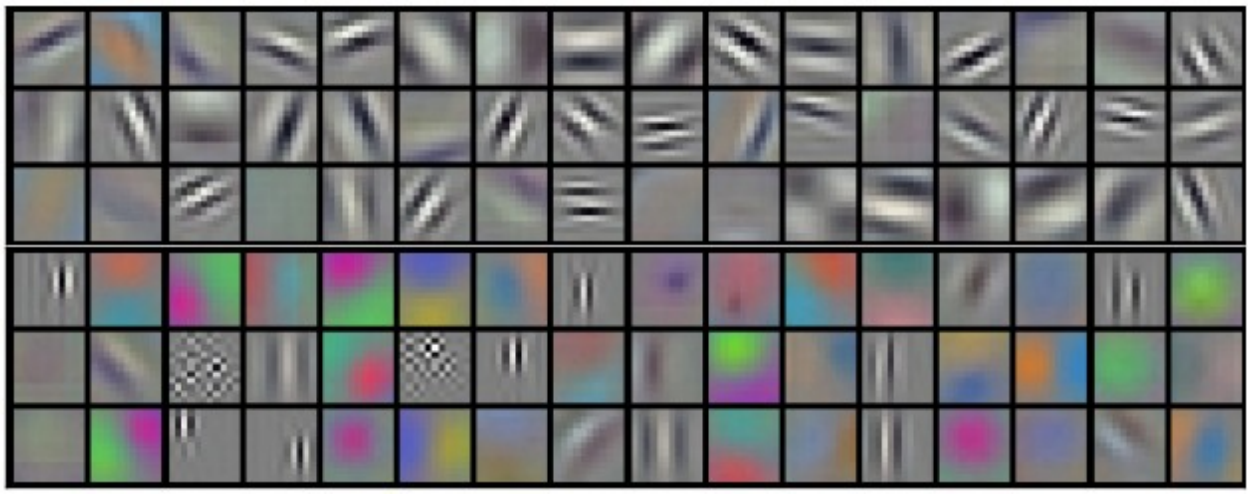

Figure 2: Example of learned filters (Krizhevsky et al. 2014)

contributes to a high activation of the filter. For the edge filter, this means that there is an edge in the image that looks similar to the edge that the filter represents.

It is common to pad the image with zeros around the edges in order for the convolution operation not to alter the spatial dimensions of the input.

5.1.1.2 Max Pooling

Max Pooling is a way to reduce the size of the feature map. Again, we slide a window (typically 2x2 pixels) across the image/activation map and produce a new, smaller map (when using 2x2 pixels: one quarter of the size). This is done by simply taking the highest value inside the sliding window as the only pixel for the new map. This causes the most prominent features of the map to persist, while less relevant information goes away. This has two benefits: it reduces computational cost, as less parameters have to be trained for the individual image maps and it helps to reduce overfitting, as the information gets more abstract.

5.1.1.3 Dropout

Dropout is another regularization technique, that works very well for various reasons. Essentially, it forces the neural net to learn multiple independent representations of the same data by alternately randomly disabling neurons in the learning phase.

5.1.1.4 Fully Connected Layer

The last (several) layer(s) of a convolutional net is a fully connected layer. Already known from the Multi-Layer Perceptron, it connects all nodes (30 in the current case) with all nodes from the previous layer.

The following image compares an abstract representation of a MLP vs. a CNN. The right side shows how the pooling operation reshapes the volume of activation maps, as the images get smaller and smaller. A fully connected layer as seen on the left side is also added in a CNN as the very last step.

5.1.2 Practice

5.1.2.1 Convenience Functions

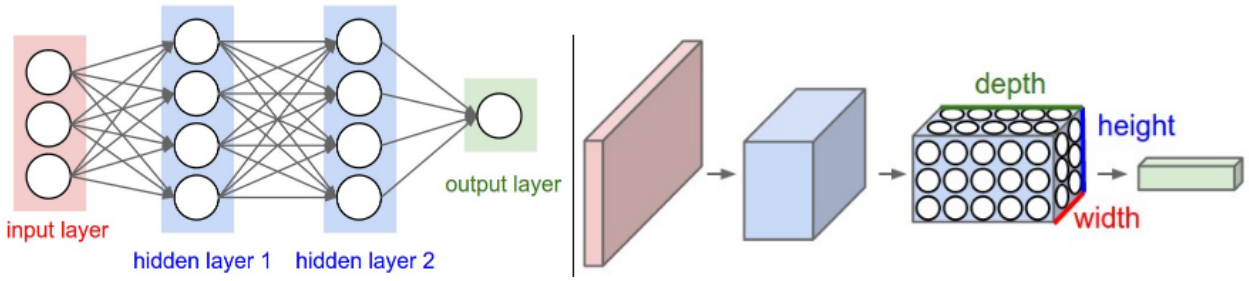

Figure 3: Andrej Karpathy, http://cs231n.github.io/convolutional-networks/ retreived on 7 Jan 2017

In addition to the *weight_variable* and *bias_variable* functions, we create functions to handle convolution and max pooling. The stride (how many pixels does the window move per step) is set to 1 for convolution, and we use 2x2 max pooling.

```
conv2d <- function(x, W) {
  tf$nn$conv2d(x, W, strides=c(1L, 1L, 1L, 1L), padding='SAME')
}

max_pool_2x2 <- function(x) {
  tf$nn$max_pool(
    x,
    ksize=c(1L, 2L, 2L, 1L),
    strides=c(1L, 2L, 2L, 1L),
    padding='SAME')
}
```

5.1.2.2 Tensorflow Graph Definition

We stack several of the layer types described in the **Theory** section on top of each other in an alternating format.

1) Convolutional layer with 32 filters and a 5x5 window size.

- Image data needs to have four-dimensional shape for internal reasons.
- Activation function is ReLU.

2) Max Pooling with 2x2 window.
3) Convolutional layer with 64 filters and a 5x5 window size.
4) Max Pooling with 2x2 window.
5) Fully connected layer with 1024 neurons to process on the entire image.

- (Apply the dropout technique for regularization).

6) Fully connected layer with 30 output neurons.

```
## First layer (convolution)
W_conv1 <- weight_variable(shape(5L, 5L, 1L, 32L))
b_conv1 <- bias_variable(shape(32L))
x_image <- tf$reshape(x, shape(-1L, 96L, 96L, 1L))
h_conv1 <- tf$nn$relu(conv2d(x_image, W_conv1) + b_conv1)
## Second layer (pooling)
h_pool1 <- max_pool_2x2(h_conv1)
## Third layer (convolution)
W_conv2 <- weight_variable(shape = shape(5L, 5L, 32L, 64L))
b_conv2 <- bias_variable(shape = shape(64L))
```

```
h_conv2 <- tf$nn$relu(conv2d(h_pool1, W_conv2) + b_conv2)
## Fourth layer (pooling)
h_pool2 <- max_pool_2x2(h_conv2)
## Fifth layer (fully connected)
W_fc1 <- weight_variable(shape(36864L, 1024L))
b_fc1 <- bias_variable(shape(1024L))
h_pool2_flat <- tf$reshape(h_pool2, shape(-1L, 24L * 24L * 64L))
h_fc1 <- tf$nn$relu(tf$matmul(h_pool2_flat, W_fc1) + b_fc1)
## Dropout
keep_prob <- tf$placeholder(tf$float32)
h_fc1_drop <- tf$nn$dropout(h_fc1, keep_prob)
## Sixth layer (readout, fully connected)
W_fc2 <- weight_variable(shape(1024L, 30L))
b_fc2 <- bias_variable(shape(30L))
y_conv <- tf$matmul(h_fc1_drop, W_fc2) + b_fc2
```

Training and evaluating the network follows the same principle as our MLP implementation.

6 Discussion and Comparison

Patch Search * Patch Search is a rather simple yet effective and intuitive approach * limitations (composition of image, limited search area (pros and cons!, search area at the borders), rotation, etc..)

7 Conclusion and Outlook