Trustless ML: An example using data obfuscation

Germán Luna

May 1, 2019

Executive Summary

In a world where so many of us want to contribute to make the world a better place, some of us find it hard knowing who to trust with our data so that they may help empower us. It is for this reason that a trustless protocol that is machine learning compatible is desired. While encryption is ideal in a scenario where Bob and Alice enjoy each other's mutual trust, data obfuscation is shown to provide another way of tackling the problem. It is shown as an example that obfuscating the MNIST dataset by permuting the rows and columns by a common permutation can be used to train a high accuracy model (96%).

Data obfuscation as a methodology

For supervised and unsupervised learning, in particular for the case of neural networks, one need in principle not know the names of the variables that are being used. Moreover, any additional information lay in the distribution of the data. What if we applied a nonlinear transformation to the data as to make the distribution of both input and output? Say we transformed them to standard multivariate normal distributions with some covariance matrix. Would machine learning still be possible?

The important aspect of such a methodology is that only the person who generated the nonlinear transformations would be able to reconstruct the original data. Of course, the difficulty at the core of such a methodology is that the nonlinear transformation be invertible, and compatible with a sufficiently diverse set of ML techniques. In this work I will be focusing on neural networks for the case of images but will discuss other areas where it can be applied.

Obfuscation for univariate and multivariate data

In order motivate the discusion lets consider the case of a single input variable X and a single output variable Y. We can use the empirical cdf's of X and Y (or some good approximation thereof) to transform $X \to X'$ and $Y \to Y'$ where X', Y' have standard normal distribution. Then the problem of learning a mapping from X' to Y' is equivalent to learning the mapping from X to Y in terms of approximate quantiles of their original distributions. Extending this approach to multivariate data can be done in a trivial way applying it to each input X_i and output variable Y_i independently, and afterwards applying a linear transformation to decorrelate the Y_i' and X_i' independently. Leaving the correlation between X' and Y' unchanged is perhaps optional, though it can be set to zero by an appropriate linear transformation. It is worth noting that because of the nonlinear transformation, the only information that is really revealed is the relative ordering of nameless data points.

Obfuscating images

It was simple enough to describe how obfuscation might work in the univariate and multivariate case when no additional structure needed to be preserved. However what about the case where we wish to preserve enough of the 2D structure of an image? It seems this would be important when, for example, one wishes to use convolutional layers. Preserving too much of the structure may fail to obfuscate the contents, but too little may significantly impair the learning process.

A relatively simple modification of the univariate case works as follows:

1. Normalize all pixels so that when looking at the same pixel across the images has a common target distribution.
2. For images, the target should be standard uniform with appropriate bounds.
3. Color channels can be transformed independently.
4. Choose a random subset of the normalized pixels and reverse their ordering while remaining distributed according to the target distribution, i.e. $x \mapsto 1 - x$.

The resulting behaviour is that constant-intensity pixels across images will map randomly (but consistently) across images to either black or white. Regions in the image where there is more variation will be thus more blurry. However, this is not sufficient for every application involving images since it is subject to some degree of decoding via a tecnique called differential cryptanalysis. In short, for images, this approach will obfuscate a single image, but will (on some datasets like MNIST) roughly preserve the differences between pairs of images (up to a sign difference) exposing the original data by considering sufficiently many of these parwise differences.

Given these observations one wonders if it's even a realizable proposition to obfuscate images meaningfully. Here I show a non-intuitive result that for some cases, like the MNIST data set, randomly shuffling the pixels by a common permutation does not significantly impair (~2-3% accuracy loss) the ability of a neural network to predict the digits *while retaining the same topology* and additionally has been noticed to produce a regularization effect that overlaps with dropout almost completely and thus reducing dropout rates increases the accuracy.

What this obfuscation approach look like on the MNIST data set

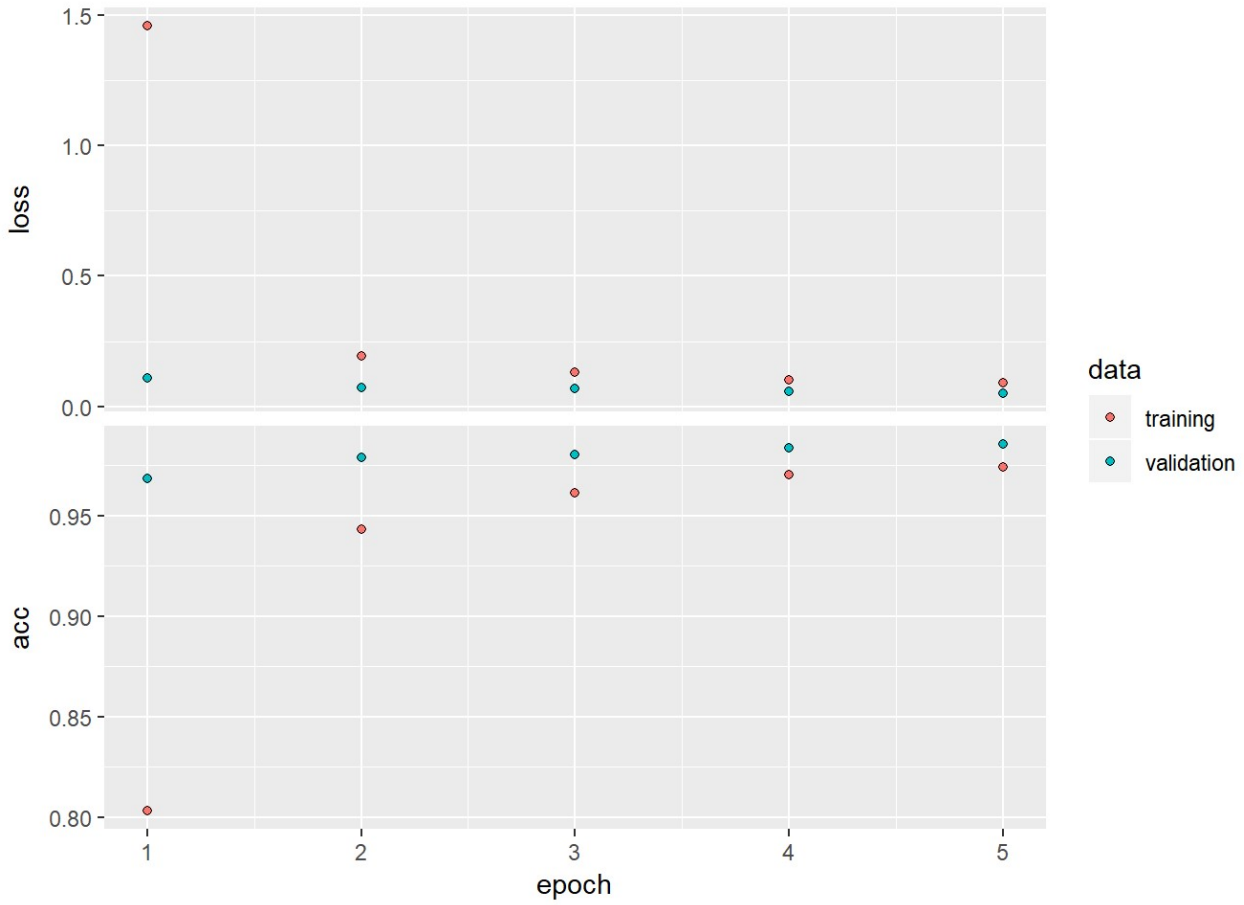
First we load the data and compute the obfuscating transformation.

```
mnist <- dataset_mnist()
x_train <- mnist$train$x
y_train <- mnist$train$y
x_test <- mnist$test$x
y_test <- mnist$test$y

nTrain <- dim(x_train)[1]
nTest <- dim(x_test)[1]
nTotal <- nTrain + nTest
all_x <- abind(x_train,x_test,along=1)
all_y <- abind(y_train,y_test,along=1)

set.seed(1)
flippedBits <- list()
shuffleIds <- sample.int(nTotal)
all_xCopy <- all_x
all_x <- all_x[shuffleIds,,]
all_y <- all_y[shuffleIds]

shuffleRows <- sample.int(28)
shuffleColumns <- sample.int(28)
all_x <- all_x[,shuffleRows,shuffleColumns]/255
all_xCopy <- all_xCopy/255

y_train_obf <- all_y[1:nTrain]
y_test_obf <- all_y[-(1:nTrain)]
x_train_obf <- all_x[1:nTrain,,]
x_test_obf <- all_x[-(1:nTrain),,]
```

Now we compare these a few pairs of images, the original (left) and the transformed (right).

As can clearly be seen, the obfuscation makes it much more difficult to rely on our usual visual cues to determine the underlying digit. Now, at this point you must be thinking that there’s no way that any useful information can be extracted from this messed up data. Before training a model on the obfuscated data we should establish a benchmark of the level of accuracy we can expect from working on the original data.

```
library(keras)
set.seed(15)

number_of_epochs <- 5
img_rows <- 28
img_cols <- 28

x_train <- array_reshape(x_train, c(nrow(x_train), img_rows, img_cols, 1))
x_test <- array_reshape(x_test, c(nrow(x_test), img_rows, img_cols, 1))
input_shape <- c(img_rows, img_cols, 1)

y_train <- to_categorical(y_train, 10)
y_test <- to_categorical(y_test, 10)


model <- keras_model_sequential() %>%
  layer_conv_2d(filters = 16, kernel_size = c(3,3), activation = 'relu',
                input_shape = input_shape) %>%
  layer_conv_2d(filters = 16, kernel_size = c(3,3), activation = 'relu') %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_dropout(rate = 0.25) %>%
  layer_flatten() %>%
  layer_dense(units = 128, activation = 'relu') %>%
  layer_dropout(rate = 0.5) %>%
  layer_dense(units = 10, activation = 'softmax')

model %>% compile(
  loss = loss_categorical_crossentropy,
  optimizer = optimizer_adadelta(),
  metrics = c('accuracy')
)

history <- model %>% fit(
  x_train, y_train,
  epochs = number_of_epochs, batch_size = 128,
  validation_split = 0.2
)

scores <- model %>% evaluate(
  x_test, y_test, verbose = 0
)
```




```
# Output metrics
cat('Train loss:', history$metrics$loss[number_of_epochs],
    '\t', 'Training accuracy: ', history$metrics$acc[number_of_epochs], '\n',
    'Validation loss:', history$metrics$val_loss[number_of_epochs],
    '\t', 'Validation accuracy: ', history$metrics$val_acc[number_of_epochs], '\n',
    'Test loss:', scores[[1]], '\t', 'Test accuracy: ', scores[[2]], '\n')
```

```
## Train loss: 0.08847872    Training accuracy:  0.97425
##  Validation loss: 0.05068516      Validation accuracy:  0.9858333
##  Test loss: 0.04351941    Test accuracy:  0.9867
```

97% on the training data and 98% accuracy is the benchmark on the test. Lets see how well we can do with the obfuscated version.

```
set.seed(15)

x_train_obf <- array_reshape(x_train_obf, c(nrow(x_train_obf), img_rows, img_cols, 1))
x_test_obf <- array_reshape(x_test_obf, c(nrow(x_test_obf), img_rows, img_cols, 1))
input_shape <- c(img_rows, img_cols, 1)
y_train_obf <- to_categorical(y_train_obf,10)
y_test_obf <- to_categorical(y_test_obf,10)

model <- keras_model_sequential() %>%
  layer_conv_2d(filters = 16, kernel_size = c(3,3),
                input_shape = input_shape) %>%
  layer_conv_2d(filters = 16, kernel_size = c(3,3), activation = 'relu') %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_dropout(rate = 0.25) %>%
  layer_flatten() %>%
  layer_dense(units = 128, activation = 'relu') %>%
  layer_dropout(rate = 0.5) %>%
  layer_dense(units = 10, activation = 'softmax')

model %>% compile(
  loss = loss_categorical_crossentropy,
  optimizer = optimizer_adadelta(),
  metrics = c('accuracy')
)

history <- model %>% fit(
  x_train_obf, y_train_obf,
  epochs = number_of_epochs, batch_size = 128,
  validation_split = 0.2
)

scores <- model %>% evaluate(
  x_test_obf, y_test_obf, verbose = 0
)
```

```
  # Output metrics
  cat('Train loss:', history$metrics$loss[number_of_epochs],
      '\t', 'Training accuracy: ', history$metrics$acc[number_of_epochs], '\n',
      'Validation loss:', history$metrics$val_loss[number_of_epochs],
      '\t', 'Validation accuracy: ', history$metrics$val_acc[number_of_epochs], '\n',
      'Test loss:', scores[[1]], '\t', 'Test accuracy: ', scores[[2]], '\n')
```

```
## Train loss: 0.1503563     Training accuracy:  0.9536458
##  Validation loss: 0.1058423   Validation accuracy:  0.9679167
##  Test loss: 0.1072125     Test accuracy:  0.9656
```

Overcoming permutations implicitly regularizes

In order for a neural network to overcome the effect of a global transformation of the input space, it doesn't necessarily have to undo that transformation. In that sense they do not care about how we choose to represent our data, only that we do so consistently. The only thing that matters is then that the architecture of the network is rich enough to express meaningful variables in the hidden layers.

In the special case of permutations of pixels considered in this work, in order for the first convolutional layer to produce filters as close as possible to the filters produced on the original data it must create sparser filters that it can then use in linear combinations at later fully connected layers. Though I do not provide the proof here, it can be shown that a convolutional layer followed by a suitably designed fully connected layer (with appropriate reshaping) applied to permuted image data can be made to have the same outputs as a convolutional layer applied to the original data. In absence of such a fully connected layer (under the assumption that there's a single best set of filters for the first layer) must learn to make do with less.

Conclusion

This work shows that there is potential to create ML models in a trustless fashion using systematically obfuscated data at the cost of a potentially small reduction in performance. Attaining 96% accuracy on MNIST using obfuscated data (as opposed to 98% on the raw data) while retaining the topology of the network identically. In addition, as a consequence of obfuscation some `regularization' is introduced as the network tries to compensate.