Machine Learning Classification with Keras:

User Guide by Claudio Casellato 861553

Overview: Classification is an important task for machine learning. It can be applied to many different scenarios where one needs to distinguish between different classes and is often used in computer vision for image recognition but its uses can be declined to many other industries, for example in the financial sector one can distinguish between value or growth stock based on financial indicators. In this guide we will build a neural network that learns to distinguish different classes sampled from different gaussian distributions with different means. The main advantage of neural networks is that they are nonlinear functions of the data and thereby capture nonlinear relations of the data resulting in a much more powerful and expressive model. We will utilize the Keras library which is part of the Tensorflow ecosystem developed by Google. To replicate the results you will need to install and import the following libraries with the following code:

```
install.packages("tensorflow")
install.packages("keras")
install.packages("tfdatasets")
install.packages("ramify")
install.packages("ggplot2")
library(keras)
library(MASS)
library(ramify)
library(ggplot2)
```

The data is sampled from different gaussians with the same covariance function and different means which lie on a circle with radius r=15. The labels for each of the point is encoded as a one-hot vector which is a vector of zeros of length of the size of the classes and has entry one where for the class it belongs to. For example if the point belongs to class 2 out of 8 classes then the vector label is [0,1,0,0,0,0,0,0]. The following is the code used to sample the training data.

```
generate_data = function(modes=6, cov=matrix(c(1,0,0,1), 2, 2), radius=15, samples=30){
  theta = seq(0, 2*pi, length.out = modes+1)
  x = cos(theta)*radius
  y = sin(theta)*radius
  mus = cbind(x,y)
  data = c()
  labels = c()
  i <- 0
  for (mu in 1:modes){
    # sample points
    sampled_data = mvrnorm(samples, mus[mu,], cov)
    data <- rbind(data, sampled_data)
    i <- i + 1
  }
  return(data)
}
generate_labels = function(modes=8, samples=30){
  labels = c()
  for (mu in 1:modes){
    # make label
    print(mu)
    labels = c(labels,replicate(samples, mu-1))
  }
  return(labels)
}
modes = 8
samples = 100
data = generate_data(modes = modes, samples = samples)
labels = generate_labels(modes = modes, samples = samples)+1
plot(data, col=colors()[labels*3], pch=15)
labels = to_categorical(labels-1)
```

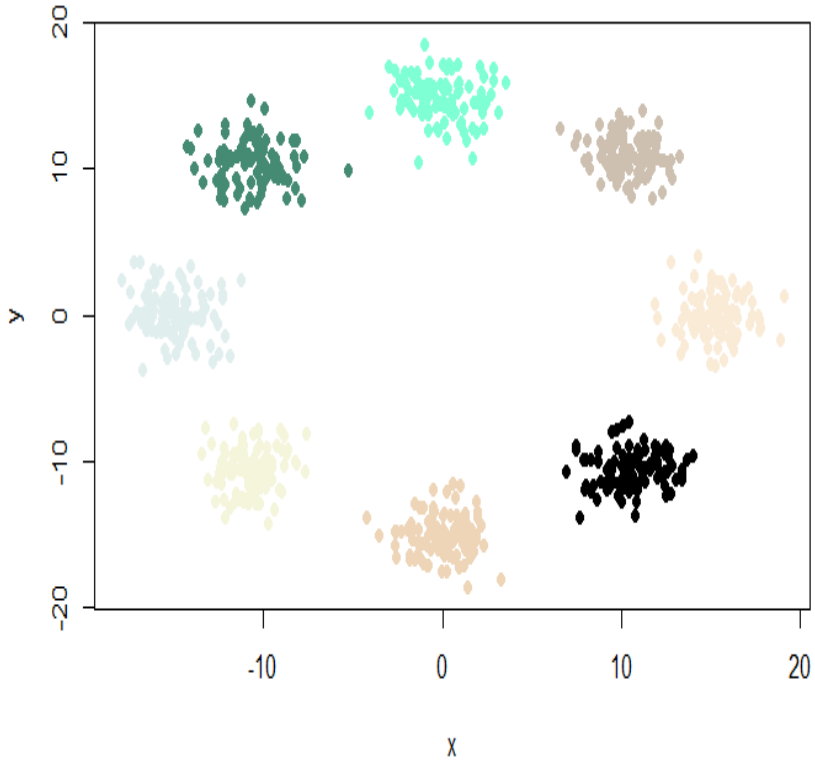


Figure 1 Sampled Data Points

Example of one-hot encoded labels for the 8 classes

> labels[c(1,150,160,300),]

	[,1]	[,2]	[,3]	[,4]	[,5]	[,6]	[,7]	[,8]
[1,]	1	0	0	0	0	0	0	0
[2,]	0	1	0	0	0	0	0	0
[3,]	0	1	0	0	0	0	0	0
[4,]	0	0	1	0	0	0	0	0

We can notice that a straight line cannot separate in any ways the different classes, therefore we need a non-linear function that can separate them. The function is a combination of linear functions and nonlinear function and the computation to achieve the final prediction are first multiply the data by a matrix of weights $W^{\mathrm{I\,x\,h}}$ where I is the input space dimension (in our case the two dimensional plane) and h is the hidden layer dimension: z = XW, where X is the data matrix where $[x^{\mathrm{t}},..]$ is a single data point. Then the vector z is passed through a nonlinear activation function $\sigma(z_{\mathrm{i}}) = \frac{1}{1+e^{-z_{\mathrm{i}}}}$ where z_{i} is each entry of the vector z. This represent the first layer of the network, which resembles the logistic regression function. The second layer computes the same operations but with z as input. The final layer is the softmax function (https://en.wikipedia.org/wiki/Softmax_function) used for classification and it represent the probability of the data point belonging to each class.

All of this is made easy with keras and to define the model one has to select the layer name from a predetermined set of layers, which in our case is the Fully-connected or Dense layer.

```
# CREATE MODEL
model <- keras_model_sequential() %>%
  layer_dense(units = 128, activation = "sigmoid") %>%
  layer_dense(units = 128, activation = "sigmoid") %>%
  layer_dense(units = modes, activation = "softmax")
```

We can check the model by the function summary(model):

> summary(model)

Model: "sequential_43"

Layer (type)	Output Shape	Param #
dense_119 (Dense)	multiple	384
dense_120 (Dense)	multiple	16512
dense_121 (Dense)	multiple	1032

Total params: 17,928

Trainable params: 17,928

Non-trainable params: 0

Then one needs to select which objective function to optimize, and in this case is the cross-entropy loss which measure the distance between two probability distributions ie. the probability distribution predicted by the model and the actual probability distribution of the data point which is the one-hot vector. Intuitively the model should put probability one on the index of the class which the data belongs to.

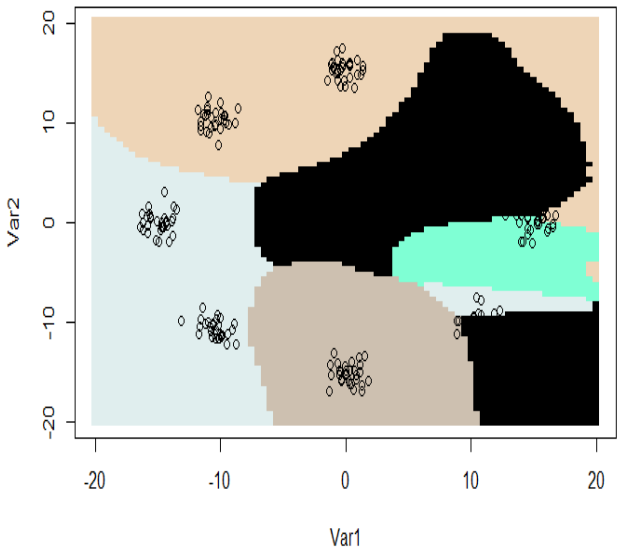
The model parameters, namely the weight matrices W are optimized through gradient descent updates. The optimizer is then selected from a set of gradient based optimizers which in our case is adam.

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

We can test the untrained model on a grid of equally spaced points to see the classification boundaries. To make predictions the the model%>%predict function is used.

```
# UNTRAINED MODEL PREDICTIONS
x = seq(-20,20,0.5)
data_test = as.matrix(expand.grid(x,x))
```

```
predictions.untrained = model%>%predict(data_test)
argmax_pred.untrained = argmax(predictions.untrained, rows=T)
plot(data_test, col=colors()[argmax_pred.untrained*3],  pch=15)
points(data)
```

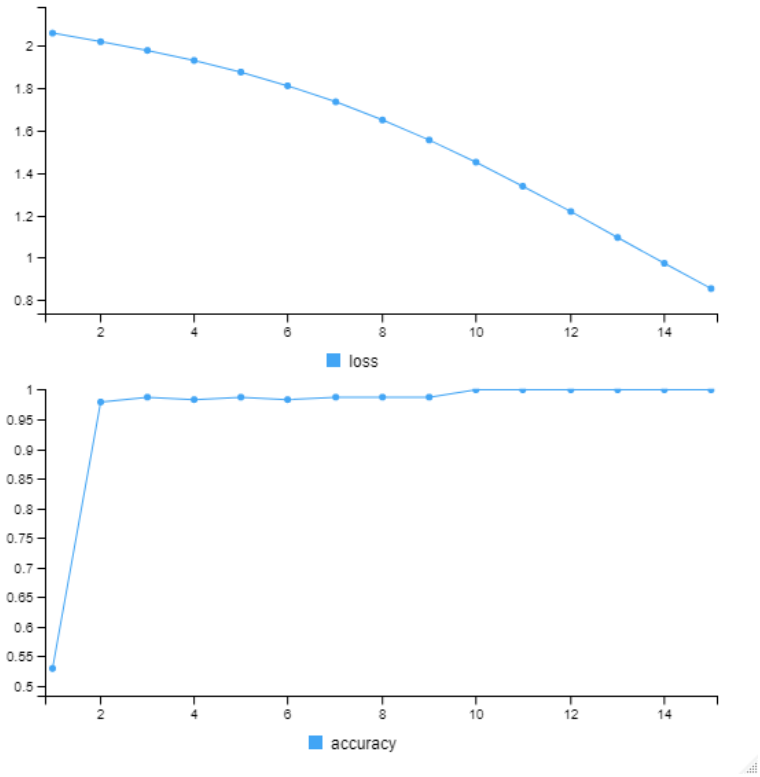


We can see that the model cannot distinguish between the different modes of the probability distribution and makes random predictions.

We then optimize the model parameters by a simple function call which is model%>%fit(data, labels,epochs) which takes the input data, the labels of the classes and the number of training iterations.

```
# TRAIN THE MODEL
model%>%fit(data, labels, epochs=15)
```

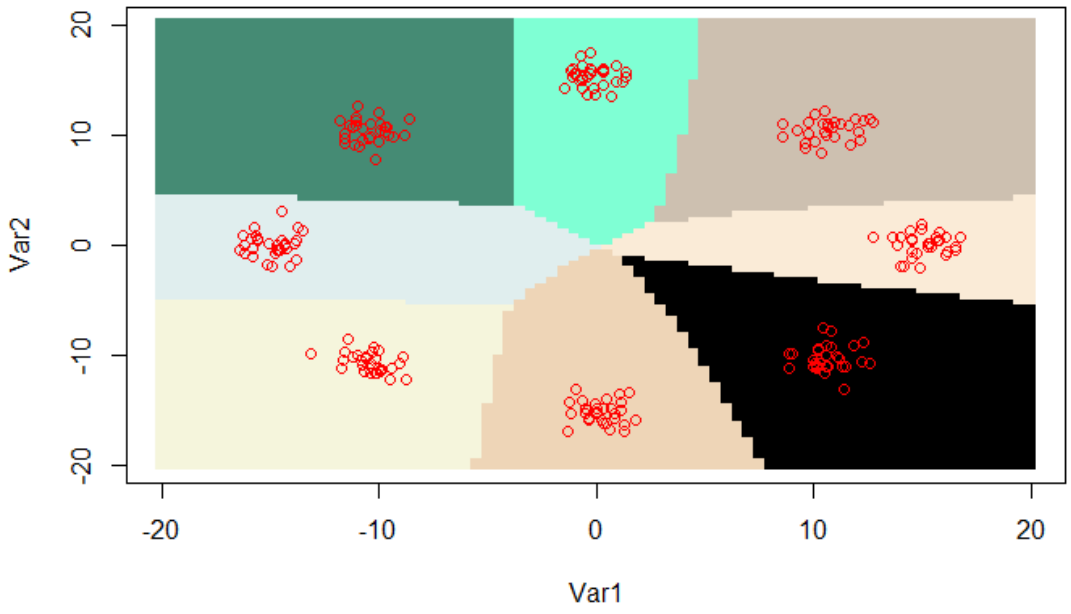
```
Train on 240 samples
Epoch 1/15
240/240 [==============================] - 1s 3ms/sample - loss: 2.0597 - accuracy: 0.5292
Epoch 2/15
240/240 [==============================] - 0s 1ms/sample - loss: 2.0191 - accuracy: 0.9792
Epoch 3/15
240/240 [==============================] - 0s 1ms/sample - loss: 1.9780 - accuracy: 0.9875
Epoch 4/15
240/240 [==============================] - 0s 2ms/sample - loss: 1.9305 - accuracy: 0.9833
Epoch 5/15
240/240 [==============================] - 0s 2ms/sample - loss: 1.8755 - accuracy: 0.9875
Epoch 6/15
240/240 [==============================] - 0s 2ms/sample - loss: 1.8108 - accuracy: 0.9833
Epoch 7/15
240/240 [==============================] - 0s 2ms/sample - loss: 1.7353 - accuracy: 0.9875
Epoch 8/15
240/240 [==============================] - 0s 2ms/sample - loss: 1.6493 - accuracy: 0.9875
Epoch 9/15
240/240 [==============================] - 0s 1ms/sample - loss: 1.5548 - accuracy: 0.9875
Epoch 10/15
240/240 [==============================] - 0s 1ms/sample - loss: 1.4503 - accuracy: 1.0000
Epoch 11/15
240/240 [==============================] - 0s 1ms/sample - loss: 1.3364 - accuracy: 1.0000
Epoch 12/15
240/240 [==============================] - 0s 2ms/sample - loss: 1.2171 - accuracy: 1.0000
Epoch 13/15
240/240 [==============================] - 0s 2ms/sample - loss: 1.0945 - accuracy: 1.0000
Epoch 14/15
240/240 [==============================] - 0s 2ms/sample - loss: 0.9726 - accuracy: 1.0000
Epoch 15/15
240/240 [==============================] - 0s 2ms/sample - loss: 0.8533 - accuracy: 1.0000
```



We can then test the new model on the regular grid to see the decision boundaries after it has been trained

```
# TEST PREDICTIONS
predictions.trained = model%>%predict(data_test)

# gets the index of the maximum probability class
argmax_pred.trained = argmax(predictions.trained, rows=T)
plot(data_test, col=colors()[(argmax_pred.trained)*3], pch=15)
points(data, col="red")
```



We can see that the model correctly segments the space into the right decision boundary. (Training points are in red for visualization porpoises)