NN_Categorical.py

```
1    ## SEIS 764: Artificial Intelligence
## Training a Neural Network
# Garth Mortensen, mort0052@stthomas.edu

# This script incorporate code and knowledge from Jason Brownloee's Machine Learning Mastery with Python
# and Datacamp.com's Deep Learning in Python

## PART 1 ---------------------------------------------------------------------
# Load packages

# Pandas- Tools and data structures to organize and analyze data
# The key to understanding Pandas for machine learning is understanding the Series and DataFrame
# data structures.

# NumPy- Allows you to efficiently work with data in arrays
# It provides the foundation data structures and operations for SciPy. These are arrays (ndarrays) that are
# efficient to define and manipulate.

# Matplotlib- Allows you to create 2D charts and plots
# *Call a plotting function with some data (e.g. .plot()).
# *Call many functions to setup the properties of the plot (e.g. labels and colors).
# *Make the plot visible (e.g. .show()).

# SKLearn Library has machine learning algorithms for classification, regression, clustering and more.
# It also has tools for related tasks such as evaluating models, tuning parameters and pre-processing.
# scikit-learn is open source and is usable commercially under the BSD license.

# Tensorflow is a more complex library for distributed numerical computation using data flow graphs.
# You can train and run very large neural networks efficiently by distributing the computations
# across potentially thousands of multi-GPU servers.
#
#  Keras is tensorflow's front-end.

# Basic math libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Machine learning libraries
from sklearn import preprocessing
from sklearn.metrics import confusion_matrix
from sklearn.metrics import roc_curve
from sklearn.metrics import auc

# Neural net libraries
import tensorflow
import keras
from keras.layers import Dense
from keras.models import Sequential
from keras.utils import to_categorical

## PART 2 ---------------------------------------------------------------------
# Load and preprocess data

# Load data
cellDNA = pd.read_csv("C:/tmp/CellDNA.csv", header=0)

# Define Y
Y = cellDNA[cellDNA.columns[13]]

# Convert to 0s and 1s
# https://stackoverflow.com/questions/38059796/can-you-use-if-else-statement-inside-braces-for-an-array-in-python
Y = np.where(Y >= 1, 1, 0)

# Convert Y to categorical for NN.
# https://stackoverflow.com/questions/44054082/keras-utils-to-categorical-name-keras-not-defined
Y = pd.DataFrame(data = Y)
Y = to_categorical(Y)
```

```
# Define X
X = cellDNA[cellDNA.columns[0:13]]

# Before standardizing X, we must convert it to float.
# This is because standardizing will add decimel points to the integers.
X = X.astype(float)

# Standardize X
X = preprocessing.scale(X, axis=0)

## PART 3 ---------------------------------------------------------------------
# Build Model

### Specify Model

# To start, you'll take the skeleton of a neural network and add a hidden layer and an output layer.
# You'll then fit that model and see Keras do the optimization so your model continually gets better.

model = Sequential()

# The question of how many neurons(Hands-On Machine Learning with Scikit-Learn & Tensorflow, pg 364)
#
# Obviously the number of neurons in the input and output layers is determined by the type of input and
# output your task requires. For example, the MNIST task requires 28 x 28 = 784 input neurons and 10
# output neurons. As for the hidden layers, a common practice is to size them to form a funnel, with fewer
# and fewer neurons at each layer – the rationale being that many low-level features can coalesce into far
# fewer high-level features. For example, a typical neural network for MNIST may have two hidden layers,
# the first with 300 neurons and the second with 100. However, this practice is not as common now, and
# you may simply use the same size for all hidden layers – for example, all hidden layers with 150
# neurons: that's just one hyperparameter to tune instead of one per layer. Just like for the number of layers,
# you can try increasing the number of neurons gradually until the network starts overfitting. In general you
# will get more bang for the buck by increasing the number of layers than the number of neurons per layer.
# Unfortunately, as you can see, finding the perfect amount of neurons is still somewhat of a black art.
# A simpler approach is to pick a model with more layers and neurons than you actually need, then use early
# stopping to prevent it from overfitting (and other regularization techniques, especially dropout, as we will
# see in Chapter 11). This has been dubbed the "stretch pants" approach: 12 instead of wasting time looking
# for pants that perfectly match your size, just use large stretch pants that will shrink down to the right size.

# The question of how many layers (Hands-On Machine Learning with Scikit-Learn & Tensorflow, pg 363)
#
# For many problems, you can just begin with a single hidden layer and you will get reasonable results. It
# has actually been shown that an MLP with just one hidden layer can model even the most complex
# functions provided it has enough neurons. For a long time, these facts convinced researchers that there
# was no need to investigate any deeper neural networks. But they overlooked the fact that deep networks
# have a much higher parameter efficiency than shallow ones: they can model complex functions using
# exponentially fewer neurons than shallow nets, making them much faster to train.
# To understand why, suppose you are asked to draw a forest using some drawing software, but you are
# forbidden to use copy/paste. You would have to draw each tree individually, branch per branch, leaf per
# leaf. If you could instead draw one leaf, copy/paste it to draw a branch, then copy/paste that branch to
# create a tree, and finally copy/paste this tree to make a forest, you would be finished in no time. Real-
# world data is often structured in such a hierarchical way and DNNs automatically take advantage of this
# fact: lower hidden layers model low-level structures (e.g., line segments of various shapes and
# orientations), intermediate hidden layers combine these low-level structures to model intermediate-level
# structures (e.g., squares, circles), and the highest hidden layers and the output layer combine these
# intermediate structures to model high-level structures (e.g., faces).

# Add the first layer
# Dense bc all nodes in previous layer connect to all nodes in current layer
model.add(Dense(10, activation='relu'))

# Add the second layer
model.add(Dense(5, activation='relu'))

# Add the output layer, model prediction
# Two categorical outcomes
model.add(Dense(2, activation='softmax'))

### Compile the Model
# To compile the model, specify the optimizer and loss function to use.
# More optimizers here https://keras.io/optimizers/#adam

#model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
# Stochastic Gradient Descent is stochastic (scientific synonym for random) bc
```

```
# mini-batches choose random observations.
# metrics accuracy because this is classification.
model.compile(loss='categorical_crossentropy', optimizer='sgd', metrics=['accuracy'])

# Verify that model contains information from compiling
print("Loss function: " + model.loss)

### Fit and Validate the model

# The data to be used as predictive features is loaded in a NumPy matrix called X
# and the data to be predicted is stored in a NumPy matrix called Y
# Default epochs = 10
#
# K-fold cross-validation is not used on deep learning models because deep learning is used
# on large datasets. To run CV on a deep net, the computational requirements would be severe.
# The computational results from a single validation run are typically trusted because
# those validation runs are reasonably large.
model.fit(X, Y, validation_split=0.3, epochs=10, batch_size=32)

### Make predictions

# Predictions will be probabilities

# Calculate predictions: predictions
# predictions = model.predict(pred_data)

# Use NumPy indexing to find the column corresponding to predicted probabilities of flagged being True.
# This is the second column (index 1) of predictions. Store the result in predicted_prob_true and print it.
# Calculate predicted probability of 1: predicted_prob_true
# predicted_prob_true = predictions[:,1]

# print predicted_prob_true
# print(predicted_prob_true)
```