Practical 1: Implementing ANNs in Keras
3rd IAPR Summer School on Document Analysis
August 2019

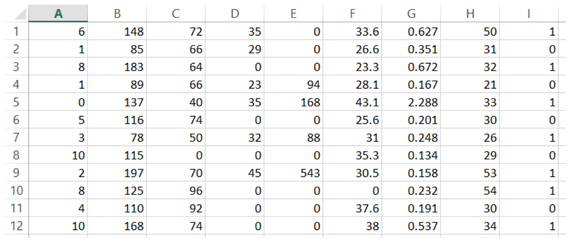
Task 1: A Two Class Problem with MLP

In this task, will discover how to create your first neural network model in Python using Keras.

The steps we will cover in this task are as follows.

- Load Data.
- Define Model.
- Compile Model.
- Fit Model.
- Evaluate Model.
- Put it all Together

In this task, we will be using the Pima Indians onset of diabetes dataset. This is a standard machine learning dataset from the UCI Machine Learning repository. It describes patient medical record data for Pima Indians and whether they had an onset of diabetes within five years. It is a binary classification problem (onset of diabetes as 1 or not as 0). All of the input variables that describe each patient are numerical. This makes it easy to use directly with neural networks. The details on different attributes of the dataset can be found here[1]. The database is provided in csv format within the practical folder and a screen shot is illustrated in Figure 1.



	A	B	C	D	E	F	G	H	I
1	6	148	72	35	0	33.6	0.627	50	1
2	1	85	66	29	0	26.6	0.351	31	0
3	8	183	64	0	0	23.3	0.672	32	1
4	1	89	66	23	94	28.1	0.167	21	0
5	0	137	40	35	168	43.1	2.288	33	1
6	5	116	74	0	0	25.6	0.201	30	0
7	3	78	50	32	88	31	0.248	26	1
8	10	115	0	0	0	35.3	0.134	29	0
9	2	197	70	45	543	30.5	0.158	53	1
10	8	125	96	0	0	0	0.232	54	1
11	4	110	92	0	0	37.6	0.191	30	0
12	10	168	74	0	0	38	0.537	34	1

Figure 1 Screen shot of PIMA Dataset

[1] https://raw.githubusercontent.com/jbrownlee/Datasets/master/pima-indians-diabetes.names

Step 1: Load Data

When working with machine learning algorithms that use a stochastic process (e.g. random numbers), it is a good idea to set the random number seed. Fixing the seed value ensures that results are reproducible.

```
from keras.models import Sequential
from keras.layers import Dense
import numpy
# fix random seed for reproducibility
numpy.random.seed(7)
```

There are eight input variables and one output variable (the last column). Once loaded we can split the dataset into input variables (X) and the output class variable (Y).

```
# load pima indians dataset
dataset = numpy.loadtxt("pima-indians-diabetes.csv", delimiter=",")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
```

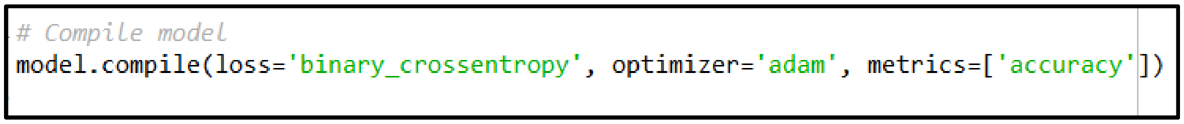
Step 2: Create Model

Models in Keras are defined as a sequence of layers. We create a Sequential model and add layers one at a time according to the desired topology. In this example, we will use a fully-connected network structure with three layers. Fully connected layers are defined using the **Dense** class. We can specify the number of neurons in the layer as the first argument, and specify the activation function using the **activation** argument. For the first layer, we also need to specify the input size with the **input_dim** argument and setting it to 8 for the 8 input variables.

```
# create model
model = Sequential()
model.add(Dense(12, input_dim=8, activation='tanh'))
model.add(Dense(8, activation='tanh'))
model.add(Dense(1, activation='sigmoid'))
```

Step 3: Compile Model

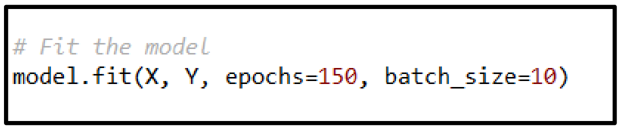
When compiling, we must specify some additional properties required when training the network. Training a network means finding the best set of weights to make predictions for this problem. We must specify the loss function to use to evaluate a set of weights, the optimizer used to search through different weights for the network and any optional metrics we would like to collect and report during training.


```
# Compile model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
```

In our example, we will use logarithmic loss, which for a binary classification problem is defined in Keras as "**binary_crossentropy**". We will also use the efficient gradient descent algorithm "**adam**" and since we are dealing with a classification problem, we will collect and report the classification accuracy as the metric.

Step 4: Fit Model

We can train or fit our model on our loaded data by calling the **fit()** function on the model. The training process will run for a fixed number of iterations through the dataset called epochs, that we must specify. We can also set the number of instances that are evaluated before a weight update in the network is performed, called the batch size and set using the **batch_size** argument. For this problem, we will run for a small number of iterations (150) and use a relatively small batch size of 10. Again, these can be chosen experimentally by trial and error.


```
# Fit the model
model.fit(X, Y, epochs=150, batch_size=10)
```

Step 5: Evaluate Model

We have trained our neural network on the entire dataset and we can evaluate the performance of the network on the same dataset. This will only give us an idea of how well we have modeled the dataset (e.g. train accuracy), but no idea of how well the algorithm might perform on new data. We have done this for simplicity, but ideally, you could separate your data into train and test datasets for training and evaluation of your model.

You can evaluate your model on your training dataset using the **evaluate()** function on your model and pass it the same input and output used to train the model.

```
# evaluate the model
scores = model.evaluate(X, Y)
print("\n%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))
```

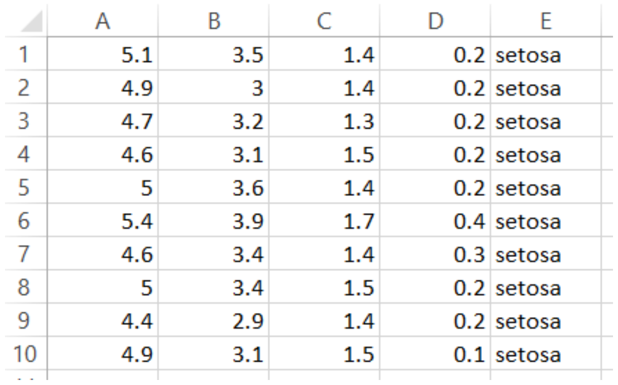
To make prediction for a new data item, you can use the function **model.predict()**.

Step 6: Putting it All Together

```
from keras.models import Sequential
from keras.layers import Dense
import numpy
# fix random seed for reproducibility
numpy.random.seed(7)

# load pima indians dataset
dataset = numpy.loadtxt("pima-indians-diabetes.csv", delimiter=",")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]

# create model
model = Sequential()
model.add(Dense(12, input_dim=8, activation='tanh'))
model.add(Dense(8, activation='tanh'))
model.add(Dense(1, activation='sigmoid'))

# Compile model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])

# Fit the model
model.fit(X, Y, epochs=150, batch_size=10)

# evaluate the model
scores = model.evaluate(X, Y)
print("\n%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))
```

Task 2: Multi-class Problem using MLP

In the second task, we will be solving a multi-class classification problem using an MLP. We will working on the famous iris dataset with 150 examples distributed into three classes. A screen shot of the data is illustrated in Figure 2.



	A	B	C	D	E
1	5.1	3.5	1.4	0.2	setosa
2	4.9	3	1.4	0.2	setosa
3	4.7	3.2	1.3	0.2	setosa
4	4.6	3.1	1.5	0.2	setosa
5	5	3.6	1.4	0.2	setosa
6	5.4	3.9	1.7	0.4	setosa
7	4.6	3.4	1.4	0.3	setosa
8	5	3.4	1.5	0.2	setosa
9	4.4	2.9	1.4	0.2	setosa
10	4.9	3.1	1.5	0.1	setosa

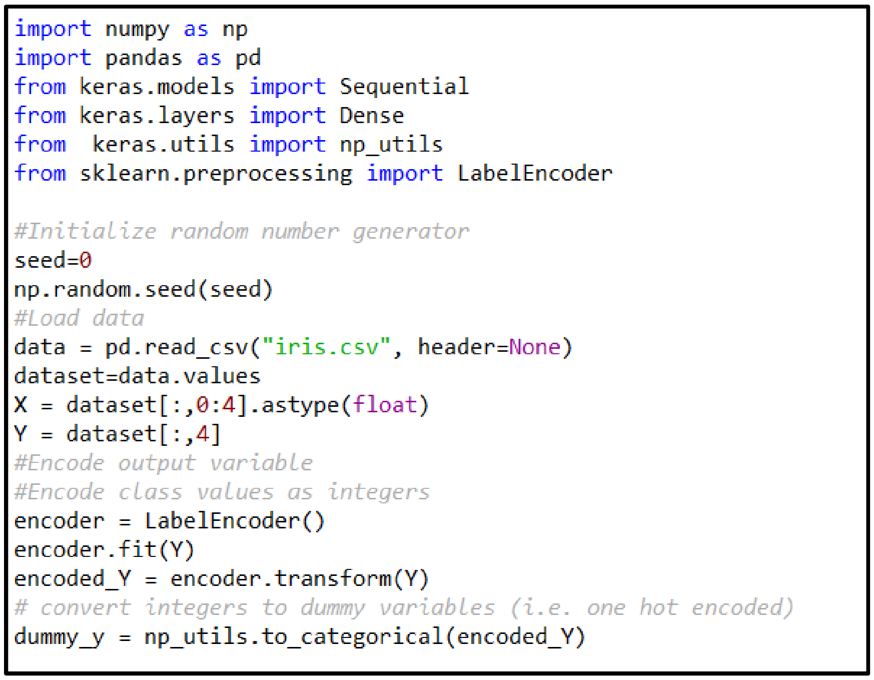
Figure 2 A screenshot of entries in the Iris Dataset

An important difference here is that the class labels are strings and not numbers. We first need to encode class labels as integers and convert the representing into one-hot-encoding prior to training the network.

```
#Encode output variable
#Encode class values as integers
encoder = LabelEncoder()
encoder.fit(Y)
encoded_Y = encoder.transform(Y)
# convert integers to dummy variables (i.e. one hot encoded)
dummy_y = np_utils.to_categorical(encoded_Y)
```

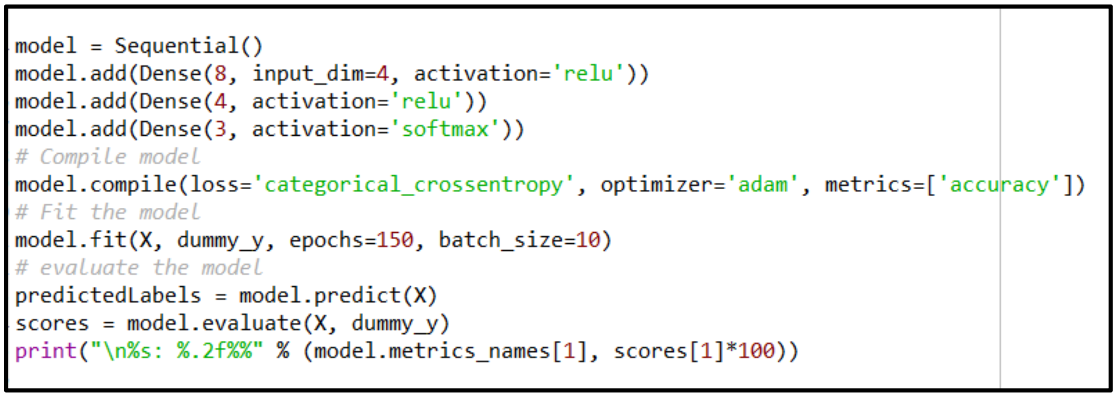
Furthermore, we will be employing the ‘softmax’ activation function at the last layer rather than the sigmoid activation function that can be employed in a two-class problem. The complete program is outlined in the following.

Loading Data and Encoding Output



```
import numpy as np
import pandas as pd
from keras.models import Sequential
from keras.layers import Dense
from  keras.utils import np_utils
from sklearn.preprocessing import LabelEncoder

#Initialize random number generator
seed=0
np.random.seed(seed)
#Load data
data = pd.read_csv("iris.csv", header=None)
dataset=data.values
X = dataset[:,0:4].astype(float)
Y = dataset[:,4]
#Encode output variable
#Encode class values as integers
encoder = LabelEncoder()
encoder.fit(Y)
encoded_Y = encoder.transform(Y)
# convert integers to dummy variables (i.e. one hot encoded)
dummy_y = np_utils.to_categorical(encoded_Y)
```

Training and Evaluation



```
model = Sequential()
model.add(Dense(8, input_dim=4, activation='relu'))
model.add(Dense(4, activation='relu'))
model.add(Dense(3, activation='softmax'))
# Compile model
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
# Fit the model
model.fit(X, dummy_y, epochs=150, batch_size=10)
# evaluate the model
predictedLabels = model.predict(X)
scores = model.evaluate(X, dummy_y)
print("\n%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))
```

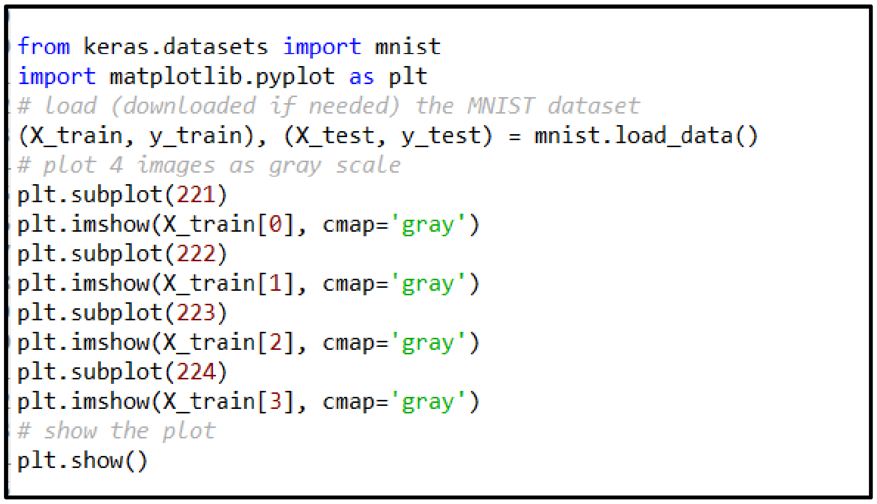
Task 3: Digit Recognition using MLP

In this task, we will be solving the classical digit recognition problem using an MLP in Keras. We will be working on the famous MNIST database developed by Yann LeCun, Corinna Cortes and Christopher Burges for evaluating machine learning models on the handwritten digit classification problem. The dataset was constructed from a number of scanned document dataset available from the National Institute of Standards and Technology (NIST). This is where the name for the dataset comes from, as the Modified NIST or MNIST dataset.

Images of digits were taken from a variety of scanned documents, normalized in size and centered. This makes it an excellent dataset for evaluating models, allowing the developer to focus on the machine learning with very little data cleaning or preparation required. Each image is a 28 by 28 pixel square (784 pixels total). A standard spit of the dataset is used to evaluate and compare models, where 60,000 images are used to train a model and a separate set of 10,000 images are used to test it.

Loading the MNIST Dataset in Keras

The Keras deep learning library provides a convenience method for loading the MNIST dataset. The dataset is downloaded automatically the first time this function is called and is stored in your home directory in ~/.keras/datasets/mnist.pkl.gz as a 15MB file. We will first write a little script to download and visualize the first 4 images in the training dataset.



```python
from keras.datasets import mnist
import matplotlib.pyplot as plt
# load (downloaded if needed) the MNIST dataset
(X_train, y_train), (X_test, y_test) = mnist.load_data()
# plot 4 images as gray scale
plt.subplot(221)
plt.imshow(X_train[0], cmap='gray')
plt.subplot(222)
plt.imshow(X_train[1], cmap='gray')
plt.subplot(223)
plt.imshow(X_train[2], cmap='gray')
plt.subplot(224)
plt.imshow(X_train[3], cmap='gray')
# show the plot
plt.show()
```

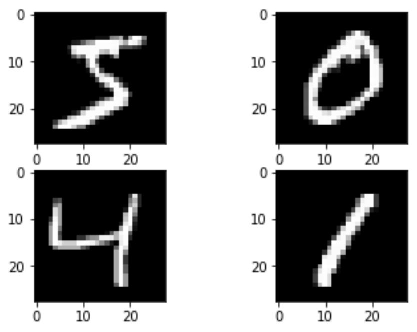


Figure 3 Sample images in the MNIST dataset

You can verify the training and test set sizes by printing the shape of each matrix.

```
print(X_train.shape)
print(y_train.shape)
print(X_test.shape)
print(y_test.shape)
```

Preparing the Data

Let's start by importing the classes and functions we need and initializing the random number generator to a constant to ensure that the results of your script are reproducible.

```
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from keras.utils import np_utils
```

```
# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
```

The training dataset is structured as a 3-dimensional array of instance, image width and image height. For a multi-layer perceptron model we must reduce the images down into a vector of pixels. In this case the 28×28 sized images will be 784 pixel input values. We can do this transform easily using the reshape() function on the NumPy array. We can also reduce our memory requirements by forcing the precision of the pixel values to be 32 bit, the default precision used by Keras anyway.

```
# flatten 28*28 images to a 784 vector for each image
num_pixels = X_train.shape[1] * X_train.shape[2]
X_train = X_train.reshape(X_train.shape[0], num_pixels).astype('float32')
X_test = X_test.reshape(X_test.shape[0], num_pixels).astype('float32')
```

The pixel values are gray scale between 0 and 255. It is almost always a good idea to perform some scaling of input values when using neural network models. Because the scale is well known and well behaved, we can very quickly normalize the pixel values to the range 0 and 1 by dividing each value by the maximum of 255.

```
# normalize inputs from 0-255 to 0-1
X_train = X_train / 255
X_test = X_test / 255
```

Finally, the output variable is an integer from 0 to 9. This is a multi-class classification problem hence we convert the labels into one hot encoding using the built-in np_utils.to_categorical() helper function in Keras.

```
# one hot encode outputs
y_train = np_utils.to_categorical(y_train)
y_test = np_utils.to_categorical(y_test)
num_classes = y_test.shape[1]
```

Training and Evaluation

You can now create the model, compile it and train it on the training data.

```
model = Sequential()
model.add(Dense(512,input_dim=num_pixels,activation='relu'))
model.add(Dense(256,activation='relu'))
model.add(Dense(10,activation='softmax'))
```

Once the model is trained, it can be evaluated on test data as follows.

```
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
model.fit(X_train,y_train,epochs=20,batch_size=10,verbose=1)
scores=model.evaluate(X_test,y_test)
print("\n%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))
```

You can also plot the evolution of accuracy and loss with respect to number of epochs.

```
history=model.fit(X_train,y_train,epochs=20,batch_size=10,verbose=1)
```

```
plt.plot(history.history['acc'])
plt.title('Model Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.show()


plt.plot(history.history['loss'])
plt.title('Model Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.show()
```

You may change the different hyper parameters and study the performance evolution.

+++++++++++