Binary Classification Sonar Project 1 for the Navy: Mines vs. Rocks

Total Marks: 100 points
Project Supervisor: Mr. Inam Ul Haq, CTO Zaavia

In this project you will discover how to effectively use the Keras library in your machine learning project by working through a binary classification project step-by-step.

After completing this project, you will know:

- How to load training data and make it available to Keras.
- How to design and train a neural network for tabular data.
- How to evaluate the performance of a neural network model in Keras on unseen data.
- How to perform data preparation to improve skill when using neural networks.
- How to tune the topology and configuration of neural networks in Keras.

Step 1. Description of the Dataset
The dataset we will use in this tutorial is the Sonar dataset.

This is a dataset that describes sonar chirp returns bouncing off different services. The 60 input variables are the strength of the returns at different angles. It is a binary classification problem that requires a model to differentiate rocks from metal cylinders.

You can learn more about this dataset on the UCI Machine Learning repository: https://archive.ics.uci.edu/ml/datasets/Connectionist+Bench+(Sonar,+Mines+vs.+Rocks)

We have download the dataset for free and placed it in the project directory with the filename sonar.csv. You can also directly download the dataset: https://archive.ics.uci.edu/ml/machine-learning-databases/undocumented/connectionist-bench/sonar/sonar.all-data

It is a well-understood dataset. All of the variables are continuous and generally in the range of 0 to 1. The output variable is a string “M” for mine and “R” for rock, which will need to be converted to integers 1 and 0.

A benefit of using this dataset is that it is a standard benchmark problem. This means that we have some idea of the expected skill of a good model. Using cross-validation, a neural network should be able to achieve performance around 84% with an upper bound on accuracy for custom models at around 88%.

Step 2. Baseline Neural Network Model Performance
Let's create a baseline model and result for this project.

We will start off by importing all of the classes and functions we will need:

```
import numpy
import pandas
from keras.models import Sequential
from keras.layers import Dense
from keras.wrappers.scikit_learn import KerasClassifier
from sklearn.model_selection import cross_val_score
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import StratifiedKFold
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
```

Next, we can initialize the random number generator to ensure that we always get the same results when executing this code. This will help if we are debugging:

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

Now we can load the dataset using pandas and split the columns into 60 input variables (X) and 1 output variable (Y). We use pandas to load the data because it easily handles strings (the output variable), whereas attempting to load the data directly using NumPy would be more difficult.

```
# load dataset
dataframe = pandas.read_csv("sonar.csv", header=None)
dataset = dataframe.values
# split into input (X) and output (Y) variables
X = dataset[:,0:60].astype(float)
Y = dataset[:,60]
```

The output variable is string values. We must convert them into integer values 0 and 1.

We can do this using the LabelEncoder class from scikit-learn. This class will model the encoding required using the entire dataset via the fit() function, then apply the encoding to create a new output variable using the transform() function.

We can use scikit-learn to evaluate the model using stratified k-fold cross validation. This is a resampling technique that will provide an estimate of the performance of the model. It does this by splitting the data into k-parts, training the model on all parts except one which is held out as a test set to evaluate the performance of the model. This process is repeated k-times

and the average score across all constructed models is used as a robust estimate of performance. It is stratified, meaning that it will look at the output values and attempt to balance the number of instances that belong to each class in the k-splits of the data.

To use Keras models with scikit-learn, we must use the KerasClassifier wrapper. This class takes a function that creates and returns our neural network model. It also takes arguments that it will pass along to the call to fit() such as the number of epochs and the batch size.

Let's start off by defining the function that creates our baseline model. Our model will have a single fully connected hidden layer with the same number of neurons as input variables. This is a good default starting point when creating neural networks.

The weights are initialized using a small Gaussian random number. The Rectifier activation function is used. The output layer contains a single neuron in order to make predictions. It uses the sigmoid activation function in order to produce a probability output in the range of 0 to 1 that can easily and automatically be converted to crisp class values.

Finally, we are using the logarithmic loss function (binary_crossentropy) during training, the preferred loss function for binary classification problems. The model also uses the efficient Adam optimization algorithm for gradient descent and accuracy metrics will be collected when the model is trained.

```
# baseline model
def create_baseline():
	# create model, write code below

	# Compile model, write code below

	return model
```

Now it is time to evaluate this model using stratified cross validation in the scikit-learn framework.

We pass the number of training epochs to the KerasClassifier, again using reasonable default values. Verbose output is also turned off given that the model will be created 10 times for the 10-fold cross validation being performed.

```
# evaluate model with standardized dataset
estimator = KerasClassifier(build_fn=create_baseline, epochs=100, batch_size=5, verbose=0)
kfold = StratifiedKFold(n_splits=10, shuffle=True, random_state=seed)
results = cross_val_score(estimator, X, encoded_Y, cv=kfold)
print("Results: %.2f%% (%.2f%%)" % (results.mean()*100, results.std()*100))
```

Running this code produces the following output showing the mean and standard deviation of the estimated accuracy of the model on unseen data.

Baseline: 81.68% (7.26%)

This is an excellent score without doing any hard work.

Step 3. Re-Run The Baseline Model With Data Preparation
It is a good practice to prepare your data before modeling.

Neural network models are especially suitable to having consistent input values, both in scale and distribution.

An effective data preparation scheme for tabular data when building neural network models is standardization. This is where the data is rescaled such that the mean value for each attribute is 0 and the standard deviation is 1. This preserves Gaussian and Gaussian-like distributions whilst normalizing the central tendencies for each attribute.

We can use scikit-learn to perform the standardization of our Sonar dataset using the StandardScaler class:
http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html

Rather than performing the standardization on the entire dataset, it is good practice to train the standardization procedure on the training data within the pass of a cross-validation run and to use the trained standardization to prepare the "unseen" test fold. This makes standardization a step in model preparation in the cross-validation process and it prevents the algorithm having knowledge of "unseen" data during evaluation, knowledge that might be passed from the data preparation scheme like a crisper distribution.

We can achieve this in scikit-learn using a Pipeline:
http://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html

The pipeline is a wrapper that executes one or more models within a pass of the cross-validation procedure. Here, we can define a pipeline with the StandardScaler followed by our neural network model.

```
# evaluate baseline model with standardized dataset
numpy.random.seed(seed)
estimators = []
estimators.append(('standardize', StandardScaler()))
estimators.append(('mlp', KerasClassifier(build_fn=create_baseline, epochs=100, batch_size=5, verbose=0)))
pipeline = Pipeline(estimators)
kfold = StratifiedKFold(n_splits=10, shuffle=True, random_state=seed)
results = cross_val_score(pipeline, X, encoded_Y, cv=kfold)
print("Standardized: %.2f%% (%.2f%%)" % (results.mean()*100, results.std()*100))
```

Running this example provides the results below. We do see a small but very nice lift in the mean accuracy.

Standardized: 84.56% (5.74%)

Step 4. Tuning Layers and Number of Neurons in The Model
There are many things to tune on a neural network, such as the weight initialization, activation functions, optimization procedure and so on.

One aspect that may have an outsized effect is the structure of the network itself called the network topology. In this section, we take a look at two experiments on the structure of the network: making it smaller and making it larger.

These are good experiments to perform when tuning a neural network on your problem.

4.1. Evaluate a Smaller Network
We suspect that there is a lot of redundancy in the input variables for this project.

The data describes the same signal from different angles. Perhaps some of those angles are more relevant than others. We can force a type of feature extraction by the network by restricting the representational space in the first hidden layer.

In this experiment, we take our baseline model with 60 neurons in the hidden layer and reduce it by half to 30. This will put pressure on the network during training to pick out the most important structure in the input data to model.

We will also standardize the data as in the previous experiment with data preparation and try to take advantage of the small lift in performance.

```
# smaller model
def create_smaller():
	# create model

	# Compile model

	return model
estimators = []
estimators.append(('standardize', StandardScaler()))
estimators.append(('mlp', KerasClassifier(build_fn=create_smaller, epochs=100, batch_size=5, verbose=0)))
pipeline = Pipeline(estimators)
kfold = StratifiedKFold(n_splits=10, shuffle=True, random_state=seed)
results = cross_val_score(pipeline, X, encoded_Y, cv=kfold)
print("Smaller: %.2f%% (%.2f%%)" % (results.mean()*100, results.std()*100))
```

Running this provides the following result. We can see that we have a very slight boost in the mean estimated accuracy and an important reduction in the standard deviation (average spread) of the accuracy scores for the model.

This is a great result because we are doing slightly better with a network half the size, which in turn takes half the time to train.

Smaller: 86.04% (4.00%)

Step 4.2. Evaluate a Larger Network

A neural network topology with more layers offers more opportunity for the network to extract key features and recombine them in useful nonlinear ways.

We can evaluate whether adding more layers to the network improves the performance easily by making another small tweak to the function used to create our model. Here, we add one new layer (one line) to the network that introduces another hidden layer with 30 neurons after the first hidden layer.

Our network now has the topology:

60 inputs -> [60 -> 30] -> 1 output

The idea here is that the network is given the opportunity to model all input variables before being bottlenecked and forced to halve the representational capacity, much like we did in the experiment above with the smaller network.

Instead of squeezing the representation of the inputs themselves, we have an additional hidden layer to aid in the process.

```
# larger model
def create_larger():
	# create model

	# Compile model

	return model
estimators = []
estimators.append(('standardize', StandardScaler()))
estimators.append(('mlp', KerasClassifier(build_fn=create_larger, epochs=100, batch_size=5, verbose=0)))
pipeline = Pipeline(estimators)
kfold = StratifiedKFold(n_splits=10, shuffle=True, random_state=seed)
results = cross_val_score(pipeline, X, encoded_Y, cv=kfold)
print("Larger: %.2f%% (%.2f%%)" % (results.mean()*100, results.std()*100))
```

Running this code produces the results below. We can see that we do not get a lift in the model performance. This may be statistical noise or a sign that further training is needed.

Larger: 83.14% (4.52%)

Step 5: Really Scaling up: developing a model that overfits
Once you've obtained a model that has statistical power, the question becomes, is your model sufficiently powerful? Does it have enough layers and parameters to properly model the problem at hand?

Remember that the universal tension in machine learning is between optimization and generalization; the ideal model is one that stands right at the border between underfitting and overfitting; between undercapacity and overcapacity. To figure out where this border lies, first you must cross it.

To figure out how big a model you'll need, you must develop a model that overfits.
This is fairly easy:

1. Add layers.
2. Make the layers bigger.
3. Train for more epochs.

Always monitor the training loss and validation loss, as well as the training and validation values for any metrics you care about. When you see that the model's performance on the validation data begins to degrade, you've achieved overfitting.

The next step is to start regularizing and tuning the model, to get as close as possible to the ideal model that neither underfits nor overfits.

Step 6: Tuning the Model
With further tuning of aspects like the optimization algorithm etc. and the number of training epochs, it is expected that further improvements are possible. What is the best score that you can achieve on this dataset?

Step 7: Rewriting the code using the Keras Functional API
Review the April 9, 2018 presentation done by Chollet contained in the project file:

Francois_Chollet_March9.pdf

Now rewrite the code that you have written so far using the Keras Sequential API in Kearas Functional API.

Step 8: Rewriting the code by doing Model Subclassing
Now rewrite the code that you have written so far using the Keras Model Subclassing as mentioned in the Chollet April 9, 2018 presentation.

Reference:
https://www.tensorflow.org/api_docs/python/tf/keras/Model
Please note you will have to use TensorFlow 1.7+ with built-in Keras.

Step 9: Rewriting the code without using scikit-learn
Once you have written the model in all three API style you are required to do k-fold cross validation without using scikit-learn library.

Step: 10: Present your Model and appear in Viva
After you have completed your project come to Sir Syed University (Lab 2) on Sunday at 4:00 pm to present your project to the project supervisor Mr. Inam ul Haq, he will also conduct a Viva. Once review of the project is complete you will be awarded marks out of 100 points.

Note: You can also consult the project supervisor on any Sunday at 4:00 pm.

What we achieved doing this Project for the Navy:
In this post, you discovered the three API styles used in Keras Deep Learning library in Python.

You learned how you can work through a binary classification problem step-by-step with Keras, specifically:

- How to load and prepare data for use in Keras.
- How to create a baseline neural network model.
- How to evaluate a Keras model using scikit-learn and stratified k-fold cross validation.
- How data preparation schemes can lift the performance of your models.
- How experiments adjusting the network topology can lift model performance.