APPLIED MACHINE LEARNING
LAB ACTIVITIES (LAB 6) (WITH SOLUTIONS)

DEEP LEARNING WITH KERAS

This workbook is designed to guide you through the activities proposed for today's lab. As you will be working independently, feel free to proceed through the text at your own pace, spending more time on the parts that are less familiar to you. The workbook contains both hands-on tasks and links to learning materials such as tutorials, articles and videos. When you are unsure about something, feel free to ask our teaching assistants or use Internet resources to look for a solution. At the end of each section, there will be questions and exercises to verify your understanding of the presented information. You may need to do some research to answer the questions.

1. Why Keras?

The TensorFlow project has adopted Keras as the high-level API for the TensorFlow 2.0 release. The biggest reasons to use Keras stem from its guiding principles, primarily the one about being user friendly. Beyond ease of learning and ease of model building, Keras offers the advantages of broad adoption, support for a wide range of production deployment options, integration with at least five backend engines (TensorFlow, CNTK, Theano, MXNet, and PlaidML), and strong support for multiple GPUs and distributed training. Plus, Keras is backed by Google, Microsoft, Amazon, Apple, Nvidia, Uber, and others.

Keras is a lightweight API and rather than providing an implementation of the required mathematical operations needed for deep learning. It provides a consistent interface to efficient numerical libraries called backends. Keras does not do its own low-level operations, such as tensor products and convolutions; it relies on a backend engine for that. Even though Keras supports multiple backend engines, its primary (and default) backend is TensorFlow, and its primary supporter is Google. The Keras API comes packaged in TensorFlow as $tf.keras$, which is the primary TensorFlow API as of TensorFlow 2.0.

In the following labs, we will mainly use Keras with TensorFlow. TensorFlow is an open source library for fast numerical computing. It was created and is maintained by Google and released under the Apache 2.0 open source license. The API is nominally for the Python programming language, although there is access to the underlying C++ API. Unlike other numerical libraries intended for use in Deep Learning like Theano, TensorFlow was designed for use both in research and development and in production systems, not least RankBrain in Google search and the fun DeepDream project. It can run on single CPU systems, GPUs as well as mobile devices and large-scale distributed systems of hundreds of machines.

Your lab PCs have both TensorFlow and Theano installed, you can configure the backend used by Keras.

2. First Deep Learning with Multi-layered Perceptron (MLP)

Load Data

Whenever we work with machine learning algorithms that use a stochastic process (e.g. random numbers), as you have been doing this, it is a good idea to initialise the random number generator with a fixed seed value. This is so that you can run the same code again and again and get the same result. This is useful if you need to demonstrate a result, compare algorithms using the same source of randomness or to debug a part of your code. You can initialise the random number generator with any seed you like, for example:

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

Verify your understanding:

(a) *Run the above code block. What is the backend engine for Keras?*

Now we can load our Pima Indians dataset. You can now load the file directly using the NumPy function $loadtxt()$. As you know the Pima Indian dataset has **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.data.csv", delimiter=",")
# split into input and output variables
X = dataset[:,0:8]
Y = dataset[:,8]
```

We are now ready to define our neural network model.

Define Model

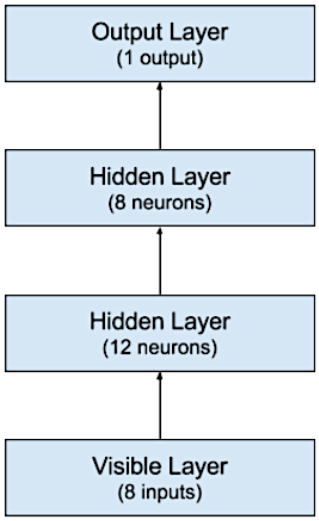
Models in Keras are defined as a sequence of layers. We create a $Sequential$ model and add layers one at a time until we are happy with our network topology. The first thing to get right is to ensure the input layer has the right number of inputs. This can be specified when creating the first layer with the input dim argument and setting it to 8 for the 8 input variables.

How do we know the number of layers to use and their types? This is a very hard question. There are heuristics that we can use and often the best network structure is found through a process of *trial and error* experimentation. Generally, you need a network large enough to capture the structure of the problem if that helps at all. 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. We will use the rectifier ($relu$) activation function on the first two layers and the sigmoid activation function in the output layer. It used to be the case that $sigmoid$ and $tanh$ activation functions were preferred for all layers. These days, better performance is seen using the $relu$ activation function. We use a $sigmoid$ activation function on the output layer to ensure our network output is between 0 and 1 and easy to map to either a probability of class 1 or snap to a hard classification of either class with a default threshold of 0.5. We can piece it all together by adding each layer. The first hidden layer has 12 neurons and expects 8 input variables (e.g. $input\ dim = 8$). The second hidden layer has 8 neurons and finally the output layer has 1 neuron to predict the class (onset of diabetes or not).

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

The figure below provides a depiction of the network structure.



Visualisation of Neural Network Structure.

Compile Model

Now that the model is defined, we can compile it. Compiling the model uses the efficient numerical libraries under the covers (i.e. backend) such as TensorFlow. The backend automatically chooses the best way to represent the network for training and making predictions to run on your hardware. When compiling, we must specify some additional properties required when training the network. Remember 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. In this case 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 $adam$ for no other reason that it is an efficient default. Learn more about the Adam optimisation algorithm in the paper *Adam: A Method for Stochastic Optimization*. See below.

https://arxiv.org/abs/1412.6980

Finally, because it is a classification problem, we will collect and report the classification accuracy as the metric.

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

Fit Model

We have defined our model and compiled it ready for efficient computation. Now it is time to execute the model on some data. 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 using the epochs argument. 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 epochs (50) 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)
```

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 modelled 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 the training and evaluation of your model.

You can evaluate your model on your training dataset using the $evaluation()$ function on your model and pass it the same input and output used to train the model. This will generate a prediction for each input and output pair and collect scores, including the average loss and any metrics you have configured, such as accuracy.

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

Verify your understanding:

(b) *Let's tie it all together and into a complete code block and run it.*

(c) What is the accuracy? Is this the accuracy we could trust? If not explain.

Soln:

Running this example, you should see a message for each of the 150 epochs printing the loss and accuracy for each, followed by the final evaluation of the trained model on the training dataset. It takes about 10 seconds to execute on my workstation running on the CPU with a Theano backend.

Data Splitting

The large amount of data and the complexity of the models require very long training times. As such, it is typical to use a simple separation of data into training and test datasets or training and validation datasets. Keras provides two convenient ways of evaluating your deep learning algorithms this way:

1. Use an automatic verification dataset.
2. Use a manual verification dataset.

Automatic Verification Dataset. Keras can separate a portion of your training data into a validation dataset and evaluate the performance of your model on that validation dataset each epoch. You can do this by setting the validation split argument on the $fit()$ function to a percentage of the size of your training dataset. For example, a reasonable value might be 0.2 or 0.33 for 20% or 33% of your training data held back for validation. The code below demonstrates the use of using an automatic validation dataset on the Pima Indians onset of diabetes dataset.

```
# MLP with automatic validation set
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.data.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='relu'))
model.add(Dense(8, activation='relu'))
model.add(Dense(1, activation='sigmoid'))

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

# Fit the model
model.fit(X, Y, validation_split=0.33, epochs=150, batch_size=10)
```

Verify your understanding:

(d) Running the above code block, you can see that the verbose output on each epoch shows the loss and accuracy on both the training dataset and the validation dataset.

Manual Verification Dataset. Keras also allows you to manually specify the dataset to use for validation during training. In this example we use the handy train test $split()$ function from the Python scikit-learn machine learning library to separate our data into a training and test dataset. We use 67% for training and the remaining 33% of the data for validation. The validation dataset can be specified to the $fit()$ function in Keras by the validation data argument. It takes a tuple of the input and output datasets.

```
# MLP with manual validation set
from keras.models import Sequential
from keras.layers import Dense
from sklearn.model_selection import train_test_split
import numpy

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

# load pima indians dataset
dataset = numpy.loadtxt("pima-indians-diabetes.data.csv", delimiter=",")

# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]

# split into 67% for train and 33% for test
X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.33,
random_state=seed)

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

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

# Fit the model
model.fit(X_train, y_train, validation_data=(X_test,y_test), epochs=150, batch_size=10)
```

Verify your understanding:

(e) Like before, running the example provides verbose output of training that includes the loss and accuracy of the model on both the training and validation datasets for each epoch.

Manual k-Fold Cross-Validation. The gold standard for machine learning model evaluation is k-fold cross-validation. It provides a robust estimate of the performance of a model on unseen data. However, cross-validation is often not used for evaluating deep learning models because of the greater computational expense. For example k-fold cross-validation is often used with 5 or 10 folds. As such, 5 or 10 models must be constructed and evaluated, greatly adding to the evaluation time of a model. Nevertheless, when the problem is small enough or if you have sufficient compute

resources, k-fold cross-validation can give you a less biased estimate of the performance of your model.

In the example below we use the handy $StratifiedKFold$ class from the scikit-learn Python machine learning library to split up the training dataset into 10 folds. See below.

https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.StratifiedKFold.html

The folds are stratified, meaning that the algorithm attempts to balance the number of instances of each class in each fold. The example creates and evaluates 10 models using the 10 splits of the data and collects all of the scores. The verbose output for each epoch is turned off by passing verbose=0 to the $fit()$ and $evaluate()$ functions on the model. The performance is printed for each model and it is stored. The average and standard deviation of the model performance is then printed at the end of the run to provide a robust estimate of model accuracy.

```
# MLP for Pima Indians Dataset with 10-fold cross validation
from keras.models import Sequential
from keras.layers import Dense
from sklearn.model_selection import StratifiedKFold
import numpy

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

# load pima indians dataset
dataset = numpy.loadtxt("pima-indians-diabetes.data.csv", delimiter=",")

# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]

# define 10-fold cross validation test harness
kfold = StratifiedKFold(n_splits=10, shuffle=True, random_state=seed)
cvscores = []
for train, test in kfold.split(X, Y):
    # create model
    model = Sequential()
    model.add(Dense(12, input_dim=8, activation='relu'))
    model.add(Dense(8, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))
    # compile model
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
    # fit the model
    model.fit(X[train], Y[train], epochs=150, batch_size=10, verbose=0)
    # evaluate the model
    scores = model.evaluate(X[test], Y[test], verbose=0)
    print("%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))
    cvscores.append(scores[1] * 100)

print("%.2f%% (+/- %.2f%%)" % (numpy.mean(cvscores), numpy.std(cvscores)))
```

Verify your understanding:

(f) Run the above code block.

(g) Why do we need a for loop?

Soln:
Notice that we had to re-create the model each loop to then fit and evaluate it with the data for the fold.

3. Use Keras with Scikit-Learn

The scikit-learn library is the most popular library for general machine learning in Python while Keras is a popular library for deep learning in Python. However the focus of the Keras library is deep learning, not all of machine learning. In fact it strives for minimalism, focusing on only what

you need to quickly and simply define and build deep learning models. The scikit-learn library in Python is built upon the SciPy stack for efficient numerical computation. It is a fully featured library for general purpose machine learning and provides many utilities that are useful in the development of deep learning models. Not least:

- Evaluation of models using resampling methods like k-fold cross-validation.
- Efficient search and evaluation of model hyperparameters.

The Keras library provides a convenient wrapper for deep learning models to be used as classification or regression estimators in scikit-learn. In this section we will work through examples of using the $KerasClassifier$ wrapper for a classification neural network created in Keras and used in the scikit-learn library.

Evaluate Models with Cross-Validation

The $KerasClassifier$ and $KerasRegressor$ classes in Keras take an argument $build\_fn$ which is the name of the function to call to create your model. You must define a function called whatever you like that defines your model, compiles it and returns it. In the example below we define a function $create\_model()$ that create a simple multilayer neural network for the problem.

We pass this function name to the $KerasClassifier$ class by the $build\_fn$ argument. We also pass in additional arguments of $epochs = 150$ and $batch\ size = 10$. These are automatically bundled up and passed on to the $fit()$ function which is called internally by the $KerasClassifier$ class. In this example we use the scikit-learn $StratifiedKFold$ to perform 10-fold stratified cross-validation. This is a resampling technique that can provide a robust estimate of the performance of a machine learning model on unseen data. We use the scikit-learn function $cross\_val\_score()$ to evaluate our model using the cross-validation scheme and print the results.

```
# MLP for Pima Indians Dataset with 10-fold cross validation via sklearn
from keras.models import Sequential
from keras.layers import Dense
from keras.wrappers.scikit_learn import KerasClassifier
from sklearn.model_selection import StratifiedKFold
from sklearn.model_selection import cross_val_score
import numpy

# create a function to build a model, required for KerasClassifier
def create_model():
    # create model
    model = Sequential()
    model.add(Dense(12, input_dim=8, activation='relu'))
    model.add(Dense(8, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))
    # compile model
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model

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

# load pima indians dataset
dataset = numpy.loadtxt("pima-indians-diabetes.data.csv", delimiter=",")

# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]

# create model
model = KerasClassifier(build_fn=create_model, epochs=150, batch_size=10, verbose=0)

# evaluate using 10-fold cross validation
kfold = StratifiedKFold(n_splits=10, shuffle=True, random_state=seed)
results = cross_val_score(model, X, Y, cv=kfold)
print(results.mean())
```

Verify your understanding:

(h) Run the above code block and explain the output.

(i) Compare the codes with the manual enumeration of cross-validation folds performed in the previous lesson.

Soln:
Running the example displays the skill of the model for each epoch. A total of 10 models are created and evaluated and the final average accuracy is displayed. You can see that when the Keras model is wrapped that estimating model accuracy can be greatly streamlined, compared to the manual enumeration of cross-validation folds performed in the previous lesson.

Grid Search Deep Learning Model Parameters

The previous example showed how easy it is to wrap your deep learning model from Keras and use it in functions from the scikit-learn library. In this example we go a step further. We already know we can provide arguments to the $fit()$ function. The function that we specify to the $build\_fn$ argument when creating the $KerasClassifier$ wrapper can also take arguments. We can use these arguments to further customise the construction of the model.

In this example we use a grid search to evaluate different configurations for our neural network model and report on the combination that provides the best estimated performance. The create $model()$ function is defined to take two arguments $optimizer$ and $init$, both of which must have default values. This will allow us to evaluate the effect of using different optimisation algorithms and weight initialisation schemes for our network. After creating our model, we define arrays of values for the parameter we wish to search, specifically:

- Optimizers for searching different weight values.
- Initializers for preparing the network weights using different schemes.
- Number of epochs for training the model for different number of exposures to the training dataset.
- Batches for varying the number of samples before weight updates.

The options are specified into a dictionary and passed to the configuration of the $GridSearchCV$ scikit-learn class. This class will evaluate a version of our neural network model for each combination of parameters ($2 \times 3 \times 3 \times 3$) for the combinations of optimizers, initializations, epochs and batches). Each combination is then evaluated using the default of 3-fold stratified cross-validation.

That is a lot of models and a lot of computation. This is not a scheme that you want to use lightly because of the time it will take to compute. It may be useful for you to design small experiments with a smaller subset of your data that will complete in a reasonable time. This experiment is reasonable in this case because of the small network and the small dataset (less than 1,000 instances and 9 attributes). Finally, the performance and combination of configurations for the best model are displayed, followed by the performance of all combinations of parameters.

```
# MLP for Pima Indians Dataset with grid search via sklearn
from keras.models import Sequential
from keras.layers import Dense
from keras.wrappers.scikit_learn import KerasClassifier
from sklearn.model_selection import GridSearchCV
import numpy

# create a function to build a model, required for KerasClassifier
def create_model(optimizer='rmsprop', init='glorot_uniform'):
    # create model
    model = Sequential()
    model.add(Dense(12, input_dim=8, kernel_initializer=init, activation='relu'))
    model.add(Dense(8, kernel_initializer=init, activation='relu'))
```

```
    model.add(Dense(1, kernel_initializer=init, activation='sigmoid'))
    # compile model
    model.compile(loss='binary_crossentropy', optimizer=optimizer, metrics=['accuracy'])
    return model

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

# load pima indians dataset
dataset = numpy.loadtxt("pima-indians-diabetes.data.csv", delimiter=",")

# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]

# create model
model = KerasClassifier(build_fn=create_model, verbose=0)

# grid search epochs, batch size and optimizer
optimizers = ['rmsprop', 'adam']
inits = ['glorot_uniform', 'normal', 'uniform']
epochs = [50, 100, 150]
batches = [5, 10, 20]
param_grid = dict(optimizer=optimizers, epochs=epochs, batch_size=batches, init=inits)
grid = GridSearchCV(estimator=model, param_grid=param_grid)
grid_result = grid.fit(X, Y)

# summarize results
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))
```

Verify your understanding:

(j) Run the above code block and interpreted the results. This might take about 5-10 minutes to complete on your workstation executed on the CPU.

Soln:
We can see that the grid search discovered that using a uniform initialisation scheme, rmsprop optimizer, 150 epochs and a batch size of 5 achieved the best cross-validation score of approximately 75% on this problem.

4. References

[1]. Géron, A., 2017. Hands-on machine learning with Scikit-Learn and TensorFlow: concepts, tools, and techniques to build intelligent systems. " O'Reilly Media, Inc.".
[2]. Raschka, S., 2015. Python machine learning. Packt Publishing Ltd.
[3]. Grus, J., 2019. Data science from scratch: first principles with python. O'Reilly Media.
[4]. Müller, A.C. and Guido, S., 2016. Introduction to machine learning with Python: a guide for data scientists. " O'Reilly Media, Inc.".
[5]. Brownlee, J., 2014. Machine learning mastery.
[6]. Raschka, S., 2015. Python machine learning. Packt Publishing Ltd.
[7]. SAS, 2018, Advanced Predictive Modelling using SAS
[8]. Géron, A., 2019. Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems. O'Reilly Media.