

Advanced* Topics in Keras

Dr. Matthew Smith, ADACS Senior Software Engineer

matthewrsmith@swin.edu.au

* Comparatively advanced 😊

Advanced Topics - Overview

- The final session of today's workshop will cover:
 - Inferencing in Keras (requiring importing and exporting models).
 - Custom Activation functions in Keras
 - Submitting training jobs to Ozstar
 - If time permits, Image Classification.

PREREQUISITES

- You can use the material contained within the git repository ADACS_ML_B as a starting point.
- Please go ahead and log into Ozstar, and move into the folder where you previously cloned ADACS_ML_B.

When we're all done, we'll move on.

INFERCING

- Inferencing is the process of:
 - Taking the learned “knowledge” of a previously trained machine, and
 - Application of this knowledge to new, previously unseen, data.
- As such, the process of inferencing might be considered as the goal of machine learning.
- Since there is no training (or heavy computation) the process of inferencing is usually quite fast.

INFRENCING

- Up to now, the codes provided to you:
 - Create a Keras model,
 - Train it – perform computations on the training data set for computing the model weights (using `model.fit()`)
 - Test it – use the test data set to check the computed weights against known data sets (using `model.evaluate()`).

INFRENCING

- To move forward, we need to:
 - Modify our existing codes to export our model in a form which can be loaded later on, and
 - Create a new code – which we shall call `infer.py` – which will perform inferencing given a single data set as an input.

EXPORTING YOUR KERAS MODEL

- Open your train.py file – the first few lines of code here should look familiar.
- The additions are the lines of code shown in the lower half.

```
# Fit the model using the training set
history = model.fit(X_train, Y_train, epochs=N_epochs, batch_size=32)

# Plot the history
plot_history(history)

# Final evaluation of the model using the Test Data
print("Evaluating Test Set")
scores = model.evaluate(X_test, Y_test, verbose=1)
print("Accuracy: %.2f%%" % (scores[1]*100))

# Export the model to file
model_json = model.to_json()
with open("model.json", "w") as json_file:
    json_file.write(model_json)
# Save the weights as well, in HDF5 format
model.save_weights("model.h5")
```

Existing

New

EXPORTING YOUR KERAS MODEL

- One method for exporting the model information to file is to employ the JSON format (pronounced JAY-SON)
- JSON: Javascript Object Notation
- This is simply a light-weight, text based format used for data exchange.
- The idea is that a JSON file is easily viewed by humans and interpreted by computer systems – it looks very similar to a C/C++ code.

```
{ "menu": {  
  "id": "file",  
  "value": "File",  
  "popup": {  
    "menuitem": [  
      { "value": "New", "onclick": "CreateNewDoc()" },  
      { "value": "Open", "onclick": "OpenDoc()" },  
      { "value": "Close", "onclick": "CloseDoc()" }  
    ]  
  }  
}}
```


EXPORTING YOUR KERAS MODEL

- The object holding the json data is produced using the code:

```
model_json = model.to_json()
```

- This is then written to file for us to load later.

```
# Fit the model using the training set
history = model.fit(X_train, Y_train, epochs=N_epochs, batch_size=32)

# Plot the history
plot_history(history)

# Final evaluation of the model using the Test Data
print("Evaluating Test Set")
scores = model.evaluate(X_test, Y_test, verbose=1)
print("Accuracy: %.2f%%" % (scores[1]*100))

# Export the model to file
model_json = model.to_json()

with open("model.json", "w") as json_file:
    json_file.write(model_json)

# Save the weights as well, in HDF5 format
model.save_weights("model.h5")
```

EXPORTING YOUR KERAS MODEL

- This is a sample of the JSON file produced by Keras using the `to_json()` function:
(at the terminal type: `cat model.json <enter>`)

```
{"class_name": "Sequential", "keras_version": "2.1.4", "config": [{"class_name": "Dense", "config": {"kernel_initializer": {"class_name": "VarianceScaling", "config": {"distribution": "uniform", "scale": 1.0, "seed": null, "mode": "fan_avg"}}, "name": "dense_1", "kernel_constraint": null, "bias_regularizer": null, "bias_constraint": null, "dtype": "float32", "activation": "relu", "trainable": true, "kernel_regularizer": null, "bias_initializer": {"class_name": "Zeros", "config": {}}, "units": 16, "batch_input_shape": [null, 128], "use_bias": true, "activity_regularizer": null}}, {"class_name": "Dense", "config": {"kernel_initializer": {"class_name": "VarianceScaling", "config": {"distribution": "uniform", "scale": 1.0, "seed": null, "mode": "fan_avg"}}, "name": "dense_2", "kernel_constraint": null, "bias_regularizer": null, "bias_constraint": null, "activation": "softmax", "trainable": true, "kernel_regularizer": null, "bias_initializer": {"class_name": "Zeros", "config": {}}, "units": 8, "use_bias": true, "activity_regularizer": null}}, {"class_name": "Dense", "config": {"kernel_initializer": {"class_name": "VarianceScaling", "config": {"distribution": "uniform", "scale": 1.0, "seed": null, "mode": "fan_avg"}}, "name": "dense_3", "kernel_constraint": null, "bias_regularizer": null, "bias_constraint": null, "activation": "softmax", "trainable": true, "kernel_regularizer": null, "bias_initializer": {"class_name": "Zeros", "config": {}}, "units": 8, "use_bias": true, "activity_regularizer": null}}]}
```

Go ahead and open the json file for a quick browse with the editor of your choice. I'll give you guys a few minutes.

EXPORTING YOUR KERAS MODEL

- In addition to this, we need to export the weights computed by the model.

`model.save_weights(INPUT)`

Where INPUT here is the name of the HDF5 file we wish to save to. This file will also be used later in infer.py.

```
# Fit the model using the training set
history = model.fit(X_train, Y_train, epochs=N_epochs, batch_size=32)

# Plot the history
plot_history(history)

# Final evaluation of the model using the Test Data
print("Evaluating Test Set")
scores = model.evaluate(X_test, Y_test, verbose=1)
print("Accuracy: %.2f%%" % (scores[1]*100))

# Export the model to file
model_json = model.to_json()
with open("model.json", "w") as json_file:
    json_file.write(model_json)

# Save the weights as well, in HDF5 format
model.save_weights("model.h5")
```

EXPORTING YOUR KERAS MODEL

- HDF5 File format = 5th generation Hierarchical Data Format (HDF), which is designed to store large amounts of data.
- Originally developed at the US's NCSA (National Center for Supercomputing Applications), home of the Blue Waters supercomputer (originally IBM, later Cray)
- We can load these files in Python (by importing the h5py module), but we won't need to do that directly.

INFER.PY – IMPORTING MODELS

IMPORTING YOUR KERAS MODEL

- Importing a previously generated Keras model is almost a simple reversal of the export steps – we need to:
 - Open the model.json file for reading and load the data,
 - Create a model from the information contained within the loaded data, and
 - Compile the model, in the same way we compiled the model during training.

IMPORTING YOUR KERAS MODEL

- Included in your cloned repository is the file you'll need – infer.py.
- This code:
 - Loads a single file (specified by the user) for classification.
 - Loads the Keras database / model, and
 - Performs a fit on that single data set.

Please open infer.py for editing.

```
GNU nano 2.3.1 File: infer.py
infer.py
# Written by Dr. Matthew Smith, Swinburne University of Technology
# Load a precomputed keras model and its weights for a single inference
# USAGE: python infer.py ID where ID is the ID of the training file
# we wish to load.

# Import modules
import numpy as np
from keras.models import Sequential
from keras.models import model_from_json
from utilities import *
import sys


# Parse the input
no_arg = len(sys.argv)
if (no_arg == 2):
    ID = int(sys.argv[1])
else:
    print("Usage: python view.py <Data_ID>")
    print("where Data_ID is a number.")
    print("Example: python infer.py 2")
    print("  -- will load X_2.dat and infer its class")
    ID = 2

print("Loading file = " + str(ID))

# We still need to know how long the time series is
N_sequence = 128      # Length of each piece of data
```

IMPORTING YOUR KERAS MODEL

- Here are the key elements of this process – the core of infer.py is here:



```
# Load the JSON file
json_file = open('model.json','r')
loaded_model_json = json_file.read()
json_file.close()

# Set up the neural layer configuration in the model
model = model_from_json(loaded_model_json)
# Load the weights into the model
model.load_weights("model.h5")
# Compile it
model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['accuracy'])
```

- In this case, the model.json and model.h5 files are in the same directory as this python script – these were produced when you executed train.py previously.

IMPORTING YOUR KERAS MODEL

- Before using the loaded weights and model, we are required to compile the model.

```
# Load the JSON file
json_file = open('model.json','r')
loaded_model_json = json_file.read()
json_file.close()

# Set up the neural layer configuration in the model
model = model_from_json(loaded_model_json)
# Load the weights into the model
model.load_weights("model.h5")
# Compile it
model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['accuracy'])
```



THE FINAL PRODUCT: INFER.PY

- Let's have a look at the entire code.
- The first new addition is the `model_from_json` module which needs to be imported.
- We can see this code parses the input provided from the command prompt – if the user does not provide an ID, we use `ID = 2` as a default value.

```
# infer.py
# Written by Dr. Matthew Smith, Swinburne University of Technol
# Load a precomputed keras model and its weights for a single i
# USAGE: python infer.py ID where ID is the ID of the training
# we wish to load.

# Import modules
import numpy as np
from keras.models import Sequential
from keras.models import model_from_json
from utilities import *
import sys

# Parse the input
no_arg = len(sys.argv)
if (no_arg == 2):
    ID = int(sys.argv[1])
else:
    print("Usage: python view.py <Data_ID>")
    print("where Data_ID is a number.")
    print("Example: python infer.py 2")
    print("  -- will load X_2.dat and infer its c")
    ID = 2

print("Loading file = " + str(ID))
```

THE FINAL PRODUCT: INFER.PY

- In this case, we are going to be lazy and load one of the test data sets for inferencing.
- Normally you would have the data you wished inferenced provided in another way – but since we are short on time, let's use the tools we have available.
- We use the `read_test_data()` function (from `utilities.py`) to load a single set of data in – if you wish to modify this later to load multiple sets, the tools are there for you.

```
# We still need to know how long the time series is
N_sequence = 128      # Length of each piece of data

# Create variables for use while inferencing.
# Keeping it in array form; you might want to inference
# multiple data sets later.
X_infer = np.empty([1,N_sequence])
Y_infer = np.empty(1)

# We can take this data from anywhere - let's load one of the training sets
X_infer[0,], Y_infer[0] = read_test_data(ID, N_sequence)
```

THE FINAL PRODUCT: INFER.PY

- After this, we are free to load our model (we could have done this first, but no matter).
- We use two functions to perform our inference:

```
# Load the JSON file
json_file = open('model.json','r')
loaded_model_json = json_file.read()
json_file.close()

# Set up the neural layer configuration in the model
model = model_from_json(loaded_model_json)
# Load the weights into the model
model.load_weights("model.h5")
# Compile it
model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['accuracy'])

# Now try classifying the single data file we loaded
Class_infer = model.predict_classes(X_infer)

# Compute the class predictions - shouldn't be used as certainties.
Class_prob = model.predict(X_infer)

print("The predicted class is %d" % Class_infer[0])
print("Class Predictions: Class 0 = %f, Class 1 = %f" % ((1.0-Class_prob[0]), Class_prob[0]))
print("The actual loaded class is %d" % Y_infer[0])
```

THE FINAL PRODUCT: INFER.PY

- The purpose of our ML engine was to predict classes – i.e. perform a classification task.
- Hence we use the `predict_classes()` function – in this case, the function will return either a `[0]` or `[1]` – if we load more data sets, it will be an array of 0's or 1's.

```
# Now try classifying the single data file we loaded
Class_infer = model.predict_classes(X_infer)
```

- There are additional inputs for the `predict_classes` function: feel free to browse the Keras documentation for these.

THE FINAL PRODUCT: INFER.PY

- The previous function returned the predicted classes, which is very convenient for us.
- If we wish the raw output of the NN to be provided, we use the `predict()` function:

```
# Compute the class predictions - shouldn't be used as certainties.  
Class_prob = model.predict(X_infer)
```

- This function replaces the `predict_proba()` function from earlier versions of Keras, though `predict_proba` should still work if used here (and provide the same result)

THE FINAL PRODUCT: INFER.PY

predict

```
predict(x, batch_size=None, verbose=0, steps=None)
```

Generates output predictions for the input samples.

Computation is done in batches.

Arguments

- **x**: The input data, as a Numpy array (or list of Numpy arrays if the model has multiple inputs).
- **batch_size**: Integer. If unspecified, it will default to 32.
- **verbose**: Verbosity mode, 0 or 1.
- **steps**: Total number of steps (batches of samples) before declaring the prediction round finished. Ignored with the default value of `None`.

THE FINAL PRODUCT: INFER.PY

- Since we have a binary classification problem with a single output in our NN, we will have a single value returned.
- It's not really a probability – but just between you and me, let's pretend it is.

```
# Load the JSON file
json_file = open('model.json','r')
loaded_model_json = json_file.read()
json_file.close()

# Set up the neural layer configuration in the model
model = model_from_json(loaded_model_json)
# Load the weights into the model
model.load_weights("model.h5")
# Compile it
model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['accuracy'])

# Now try classifying the single data file we loaded
Class_infer = model.predict_classes(X_infer)

# Compute the class predictions - shouldn't be used as certainties.
Class_prob = model.predict(X_infer)

print("The predicted class is %d" % Class_infer[0])
print("Class Predictions: Class 0 = %f, Class 1 = %f" % ((1.0-Class_prob[0]), Class_prob[0]))
print("The actual loaded class is %d" % Y_infer[0])
```


THE FINAL PRODUCT: INFER.PY

- Since we are using the functions contained within utilities.py, and loading the test data for inferencing, we should place infer.py in the same directories as train.py.
- When we call this script (python infer.py 24), this is what we get:

```
Loading file = 24
Loading file ./Test/X_24.dat
The predicted class is 0
Class Predictions: Class 0 = 0.931161, Class 1 = 0.068839
The actual loaded class is 0
```

- You can see that we are quite sure that the class is not class 1 (accurate), and we have correctly picked the class.

ACTIVITY

- Run `train.py` once and ensure that the json and model files are there.
- Run `infer.py` `python infer.py 5 <enter>` (for example)
- Check to make sure the predicted and actual classification matches.

Let us know when you are done – then we may move on.

CUSTOM ACTIVATION FUNCTIONS

CUSTOM ACTIVATION FUNCTIONS

- In your previous experimentation, you will have noticed that the choice of activation functions within each layer does influence the performance of the model.
- Research into new activation functions is very active, and on-going.
- While Keras will continue to be supported with newer functions being added, it is likely you will encounter situations where you might want to add your own function.

CUSTOM ACTIVATION FUNCTIONS

- Today we are going to implement a recently proposed activation function developed by Google – the swish activation function.

SEARCHING FOR ACTIVATION FUNCTIONS

Prajit Ramachandran*, Barret Zoph, Quoc V. Le
Google Brain
{prajit, barretzoph, qvl}@google.com

ABSTRACT

The choice of activation functions in deep networks has a significant effect on the training dynamics and task performance. Currently, the most successful widely-used activation function is the Rectified Linear Unit (ReLU). Although various hand-designed alternatives to ReLU have been proposed, none have managed to replace it due to inconsistent gains. In this work, we propose to

Is it Time to Swish? Comparing Deep Learning Activation Functions Across NLP tasks

Steffen Eger, Paul Youssef, Iryna Gurevych
Ubiquitous Knowledge Processing Lab (UKP-TUDA)
Department of Computer Science
Technische Universität Darmstadt
www.ukp.tu-darmstadt.de

Abstract

Activation functions play a crucial role in neural networks because they are the non-

ReLU function (Glorot et al., 2011) has proven much more suitable. It has an identity derivative in the positive region and is thus claimed to

SWISH ACTIVATION FUNCTION

- I'll be using the swish function of the form:

$$\text{swish}(x) = \beta x \text{ sigmoid}(x)$$

where Beta is a constant, x is our input vector and sigmoid is:

$$\text{sigmoid}(x) = \frac{\exp(x)}{\exp(x) + 1}$$

SWISH ACTIVATION FUNCTION

- Let's write some code – open utilities.py and make some changes:
 - We'll need the keras backend if we want to borrow the sigmoid function – so import keras.
 - Define a new function (swish(x)) for us to use in our model.

```
# Utilities.py
# Dr. Matthew Smith, Swinburne University of Technology
# Various tools prepared for the ADACS Machine Learning workshop

# Import modules
import matplotlib
matplotlib.use('tkagg')
import matplotlib.pyplot as plt
import numpy as np
import keras

def swish(x):
    # Swish activation function
    beta = 1.5
    return beta*x*keras.backend.sigmoid(x)
```

Feel free to use numpy's `exp()` function if you want to write out the sigmoid function explicitly and avoid importing Keras or using its backend.

Please make the changes to the utilities.py code.

SWISH ACTIVATION FUNCTION

- To use it, we need to edit the train.py file
 - go ahead and open it for editing.
- Replace the activation function on the input or hidden layer with your swish() function.

```
# Create our Keras model - an RNN (in Keras this is a Sequence)
model = Sequential()

# Let's add some dropout on the input layer.
# We'll duplicate the input dimension to make it easier to comment out
model.add(Dropout(0.5, input_shape=(N_sequence,)))
model.add(Dense(16, activation='relu', input_dim=N_sequence))
model.add(Dense(8, activation = swish)) ←
#model.add(Dense(8, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
```

Please make the changes to the train.py code, and run it to make sure it works.

NOTE ON INFERENCE...

- When you run this `train.py` file, you will produce the `json` and `h5` files you need to perform inference.
- However, if you attempt now to run `infer.py` ..Well, I think the best thing to do is try it out.

Run your `infer.py` script and see what happens.

NOTE ON INFERENCE...

- It shouldn't have worked.
- The swish function is not part of the standard keras library. Even though the swish function is def'd in utilities.py, an error appears.

```
Traceback (most recent call last):
  File "infer.py", line 46, in <module>
    model = model_from_json(loaded_model_json)
  File "build/bdist.linux-x86_64/egg/keras/models.py", line 34
  File "build/bdist.linux-x86_64/egg/keras/layers/__init__.py"
  File "build/bdist.linux-x86_64/egg/keras/utils/generic_utils
  File "build/bdist.linux-x86_64/egg/keras/models.py", line 13
  File "build/bdist.linux-x86_64/egg/keras/layers/__init__.py"
  File "build/bdist.linux-x86_64/egg/keras/utils/generic_utils
  File "build/bdist.linux-x86_64/egg/keras/engine/topology.py"
  File "build/bdist.linux-x86_64/egg/keras/legacy/interfaces.p
  File "build/bdist.linux-x86_64/egg/keras/layers/core.py", li
  File "build/bdist.linux-x86_64/egg/keras/activations.py", li
  File "build/bdist.linux-x86_64/egg/keras/activations.py", li
  File "build/bdist.linux-x86_64/egg/keras/utils/generic_utils
ValueError: Unknown activation function:swish
```

NOTE ON INFERENCE...

- We need to inform keras that we are using a custom function.
- Please open infer.py for editing, and make the following change:

```
# Set up the neural layer configuration in the model
model = model_from_json(loaded_model_json, custom_objects={'swish':swish})
#model = model_from_json(loaded_model_json)
# Load the weights into the model
model.load_weights("model.h5")
# Compile it
model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['accuracy'])
```

Run your infer.py script again – it should work now.

ACTIVITY

- The Penalised Tanh activation function has recently been demonstrated to provide consistently strong performance in RNN's.
- Modify your code to use the Penalised Tanh activation function:

$$f(x) := \begin{cases} \tanh(x), & x > 0 \\ 0.25 \tanh(x), & x \leq 0 \end{cases}$$

- Apply this, together with dropout and filtering, on your previously modified data set for training.
- Test your infer.py with this new function as well.

IMAGE CLASSIFICATION WITH KERAS

IMAGE CLASSIFICATION

- Another popular application of RNN's is for the use of Image Classification.
- There are several concepts we will need to investigate before we can practically do this:
 - Data Generators
 - Data Augmentation
 - Convolution, Pooling and Flattening
- After that, the rest of the code is virtually unchanged.

PROBLEM INTRODUCTION

- In this case, I have prepared a set of images for us to use for a binary classification problem – a group of woofs and meows taken from the Kraggle Cats and Dogs dataset (I've got the images ready for you on /fred)



These images are all of different dimensions (i.e. numbers of pixels) with varying amounts of noise.

PROBLEM INTRODUCTION

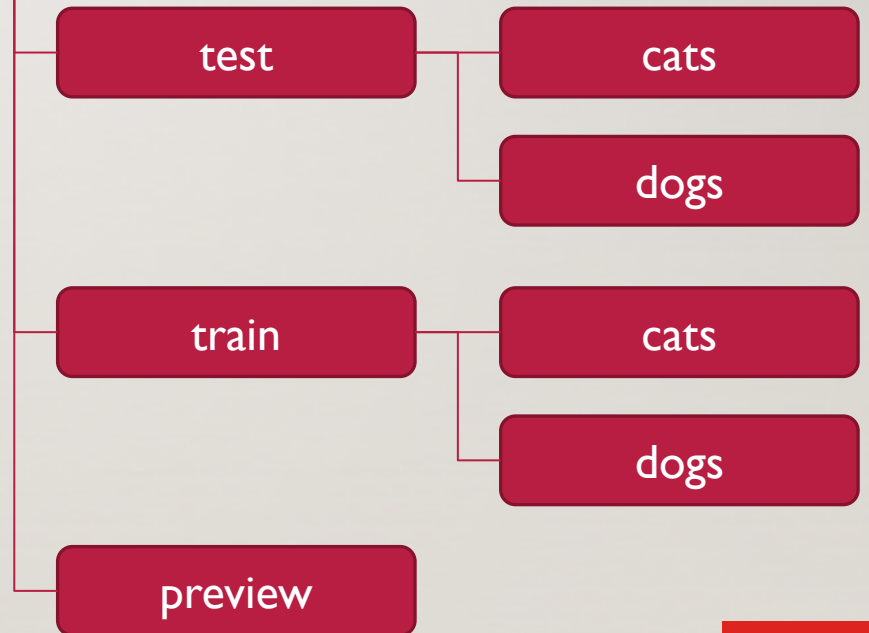
DIRECTORY

- You can git clone the scripts you'll need through this repository.

```
git clone https://github.com/archembaud/ADACS_ML_D
```

AND/OR you can pull it from /fred/oz989/IMAGE_DATA
- all of the files you'll need for this are there.

- Copy it into a directory in your home folder – you'll also need to set up these directories:



Set up your files / folders now

PROBLEM INTRODUCTION

DIRECTORY

- Each one of the 4 training and test folders – cats and dogs respectively – ought to hold the JPG files used for this demonstration.
- You'll need to move the tar.gz files into the correct locations, and unzip them.
- There are 1000 training images (cats and dogs, 2000 total) and 100 test images (200 total) – 2200 images all together.



Let's get this down now, please.

DATA GENERATORS

- The amount of data involved with image classification can be quite large – in this case, we only have 2400 images (2000 train, 200 test) but even so, the amount of data is quite large compared to previous examples.
- Best practice in this case is to make use of data generators – this allows us to load data in parallel (using multiple CPU cores) along with other advantages (related to memory use).

DATA GENERATORS

- Using generators typically results in two major differences when compared to the previously covered examples:
 - We do not manually load the whole data set in. We need to set up the generators to load data when the data is required, which happens in batches.
 - We cannot use the `fit()` and `evaluate()` functions as we did before – since the whole dataset is never on hand. We instead use `fit_generator()` and `evaluate_generator()` functions.

DATA GENERATORS

- Let's get into it – please open train.py for editing using whatever editor you like.
- We are looking for the train_datagen and test_datagen types, just a little down from the top.

```
GNU nano 2.3.1 File:
# Image classification demonstration with Keras
# Dr. Matthew Smith, Swinburne University of Technology, CAS / ADACS
# Before using this script, make sure you:
# (i) Create the train and test directories properly.
# You'll need: ./train/cats, ./train/dogs, ./test/cats, ./test/dogs, ./preview
# (ii) Extract the zip files into the correct location
# There will be 1000 training images each of cats and dogs, with 100 training images a
# The preview folder, initially empty, will be filled by the Preview_Image_Generator()

from keras.preprocessing.image import ImageDataGenerator
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D
from keras.layers import Activation, Dropout, Flatten, Dense
from keras import backend as K
from utilities import *

# Investigate the imagedatagenerator
# This is not part of the actual ML process - you can
# comment this out after its first use
Preview_Image_Generator(True)

# Dimensions of our images.
img_width, img_height = 150, 150

train_dir = './train'
test_dir = './test'
N_train = 2000 # Total number of training files we have (1000+1000)
N_test = 200 # Total number of test files we have (100+100)
N_epochs = 1 # This is not going to be enough.

# Prepare augmented training data generator
train_datagen = Prepare_Image_Data(0.2, 0.2, True, False)
test_datagen = Prepare_Image_Data(0.0, 0.0, False, True)

# Create the training and testing data generators.
train_generator = train_datagen.flow_from_directory(train_dir,
                                                    target_size=(img_width, img_height), batch_size=32, c
```

DATA GENERATORS

- First, we create our `train_datagen` and `test_datagen` structures using the `Prepare_Image_Data()` functions (inside `utilities.py`) – back to this soon.

```
# Prepare augmented training data generator
train_datagen = Prepare_Image_Data(0.2, 0.2, True, False)
test_datagen = Prepare_Image_Data(0.0, 0.0, False, True)

# Create the training and testing data generators.
train_generator = train_datagen.flow_from_directory(train_dir,
                                                    target_size=(img_width, img_height), batch_size=32, class_mode='binary')

test_generator = test_datagen.flow_from_directory(test_dir,
                                                  target_size=(img_width, img_height), batch_size=32, class_mode='binary')
```

- What is more important is the 2nd group of commands, and the batch size.

DATA GENERATORS

- The batch size is important because, later on when performing training and evaluating, data will be loaded into memory in groups of 32 (in this case).

```
# Prepare augmented training data generator
train_datagen = Prepare_Image_Data(0.2, 0.2, True, False)
test_datagen = Prepare_Image_Data(0.0, 0.0, False, True)

# Create the training and testing data generators.
train_generator = train_datagen.flow_from_directory(train_dir,
                                                    target_size=(img_width, img_height), batch_size=32, class_mode='binary')

test_generator = test_datagen.flow_from_directory(test_dir,
                                                  target_size=(img_width, img_height), batch_size=32, class_mode='binary')
```

- This means that we will have $(2000//32 = 62)$ batches to run through.

CONVOLUTION

- The process of convolution in neural networks is not related to mathematical convolution. It's very misleading.
- If time permits, I'll talk to you a little about the convolution process.
- Practically, it has a lot in common with the moving filter we applied on the ID data previously.

CONVOLUTION

- You can find more info about convolutional layers here:

<https://keras.io/layers/convolutional/>

- Convolution provides volumes of data, depending on our parameters, which needs to be flattened before we feed it into a normal NN.

```
# Create the model
model = Sequential()

# Add convolution and pooling layers to find features and reduce problem size.
model.add(Conv2D(32, (3, 3), input_shape=input_shape, activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(32, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

# Reduce our problem to a one dimensional form
# After this, it is like we have a 1D sequence
# as we did in previous examples.
model.add(Flatten())

# From here it is business as usual.
# This is a binary classification problem, so make sure the output layer
# has one output. Don't place dropout on the output layer.
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.6))
model.add(Dense(1, activation='sigmoid'))
```


OZSTAR SUBMISSIONS

- Against my better judgement, try running the script from the head node:

```
python train.py <enter>
```

- It's only 1 epoch, but it still takes quite a lot of time. It would be pretty irresponsible to run this on the head (log-in) node of Ozstar.
- We need to submit a job to the Ozstar queue using sbatch.

OZSTAR SUBMISSIONS

- There are many good examples here:

<https://supercomputing.swin.edu.au>

- Here is a script which would be suitable for today.
- There are only a couple of lines which might need special attention.
- This is not a tutorial on Ozstar job submission – but if you have no idea where to start, copy this.

jobscript.sh (found in the last repository you cloned)

```
#!/bin/bash
#
#SBATCH --job-name=Train_Py
#SBATCH --output=log.txt
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=1
#SBATCH --ntasks-per-node=1
#SBATCH --time=1:00:00
#SBATCH --mem-per-cpu=1000
#SBATCH --partition=skylake
#SBATCH --gres=gpu:1
#SBATCH --account=oz989
#SBATCH --reservation=m1

# Load the modules
module load numpy/1.14.1-python-2.7.14
module load tensorflowgpu/1.6.0-python-2.7.14
module load scikit-learn/0.19.1-python-2.7.14
module load keras/2.1.4-python-2.7.14
module load h5py/2.7.1-python-2.7.14-serial


# Run the script
python train.py
```

OZSTAR SUBMISSIONS

- Note: You automatically made yourself a copy of this when you git cloned the repository. Edit it with me.
- We have a couple of nodes especially allocated to us today for the purpose of the workshop – we gain access to these through --reservation:

```
#SBATCH --reservation=m1
```

- These nodes will be deactivated (SOON!) so, in the future, don't use this.



```
#!/bin/bash
#
#SBATCH --job-name=Train_Py
#SBATCH --output=log.txt
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=1
#SBATCH --ntasks-per-node=1
#SBATCH --time=1:00:00
#SBATCH --mem-per-cpu=1000
#SBATCH --partition=skylake
#SBATCH --gres=gpu:1
#SBATCH --account=oz989
#SBATCH --reservation=m1

# Load the modules
module load numpy/1.14.1-python-2.7.14
module load tensorflowgpu/1.6.0-python-2.7.14
module load scikit-learn/0.19.1-python-2.7.14
module load keras/2.1.4-python-2.7.14
module load h5py/2.7.1-python-2.7.14-serial

# Run the script
python train.py
```

OZSTAR SUBMISSIONS

Note the log.txt file for output here.

- In the future, you'll also need to run jobs through your own Ozstar project accounts. So the following line:

```
#SBATCH -account=oz989
```

will also need to be replaced.

- Don't forget to ask for a GPU since we are using the tensorflowgpu module..

```
#!/bin/bash
#
#SBATCH --job-name=Train_Py
#SBATCH --output=log.txt
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=1
#SBATCH --ntasks-per-node=1
#SBATCH --time=1:00:00
#SBATCH --mem-per-cpu=1000
#SBATCH --partition=skylake
#SBATCH --gres=gpu:1
#SBATCH --account=oz989
#SBATCH --reservation=m1

# Load the modules
module load numpy/1.14.1-python-2.7.14
module load tensorflowgpu/1.6.0-python-2.7.14
module load scikit-learn/0.19.1-python-2.7.14
module load keras/2.1.4-python-2.7.14
module load h5py/2.7.1-python-2.7.14-serial

# Run the script
python train.py
```

OZSTAR SUBMISSIONS

- Use sbatch to submit the job:

```
sbatch jobscript.sh <enter>
```

- You'll get a message saying the job has been submitted, and a batch job number.

```
Username@farnarkle1 ImageClassification]$ sbatch jobscript.sh  
Submitted batch job 2415377
```

- The regular text / information which appears on the screen when you previously ran the jobs on the headnode will be saved to **log.txt**

OZSTAR SUBMISSIONS

- Edit the job.txt
- As the job goes on, more text will appear in this file. If it is empty, it is likely because your job is still running – or hasn't started yet.
- The GPU will only allow one user access at any time – there are a few GPU's on this node, but still – if the reserved nodes are full, your job will spill over into the regular Ozstar cue.

```
GNU nano 2.3.1 File: log.txt
-----
A process has executed an operation involving a call to the
"fork()" system call to create a child process. Open MPI is currently
operating in a condition that could result in memory corruption or
other system errors; your job may hang, crash, or produce silent
data corruption. The use of fork() (or system() or other calls that
create child processes) is strongly discouraged.

The process that invoked fork was:

Local host:          [[63984,1],0] (PID 342820)

If you are *absolutely sure* that your application will successfully
and correctly survive a call to fork(), you may disable this warning
by setting the mpi_warn_on_fork MCA parameter to 0.
-----
/apps/skylake/software/mpi/gcc/6.4.0/openmpi/3.0.0/h5py/2.7.1-python-2.7.14/lib/python2.7/s
from ._conv import register_converters as _register_converters
Using Theano backend.
There will be 62 training groups here
Found 2000 images belonging to 2 classes.
Found 200 images belonging to 2 classes.

-----
Layer (type)                Output Shape              Param #
-----
conv2d_1 (Conv2D)           (None, 148, 148, 32)      896
-----
max_pooling2d_1 (MaxPooling2 (None, 74, 74, 32)        0
-----
conv2d_2 (Conv2D)           (None, 72, 72, 32)       9248
-----
max_pooling2d_2 (MaxPooling2 (None, 36, 36, 32)        0
-----
conv2d_3 (Conv2D)           (None, 34, 34, 64)       18496
-----
max_pooling2d_3 (MaxPooling2 (None, 17, 17, 64)        0
-----
flatten_1 (Flatten)         (None, 18496)             0
-----
dense_1 (Dense)             (None, 64)                1183808
-----
```

WRAPPING UP

- You'll need to save the models and weights for job submissions on Ozstar to be of any use. Otherwise, apart from the log.txt file, its all lost.
- I haven't shown you how to export and load the model required for image classification – it is very similar to the previously listed examples; I'm sure you'll figure things out on your own.

WRAPPING UP

- Tomorrow:
 - You'll be working alone (in your groups) to solve the LIGO signal classification problem. This will be shared with you tomorrow.
 - You can also work on your own problems, or modify the image classifier I've presented here for galaxy classification (for example).
 - You will absolutely need access to `/fred/oz989` to get the LIGO data, but more info for this will be shown tomorrow morning.

See you tomorrow.