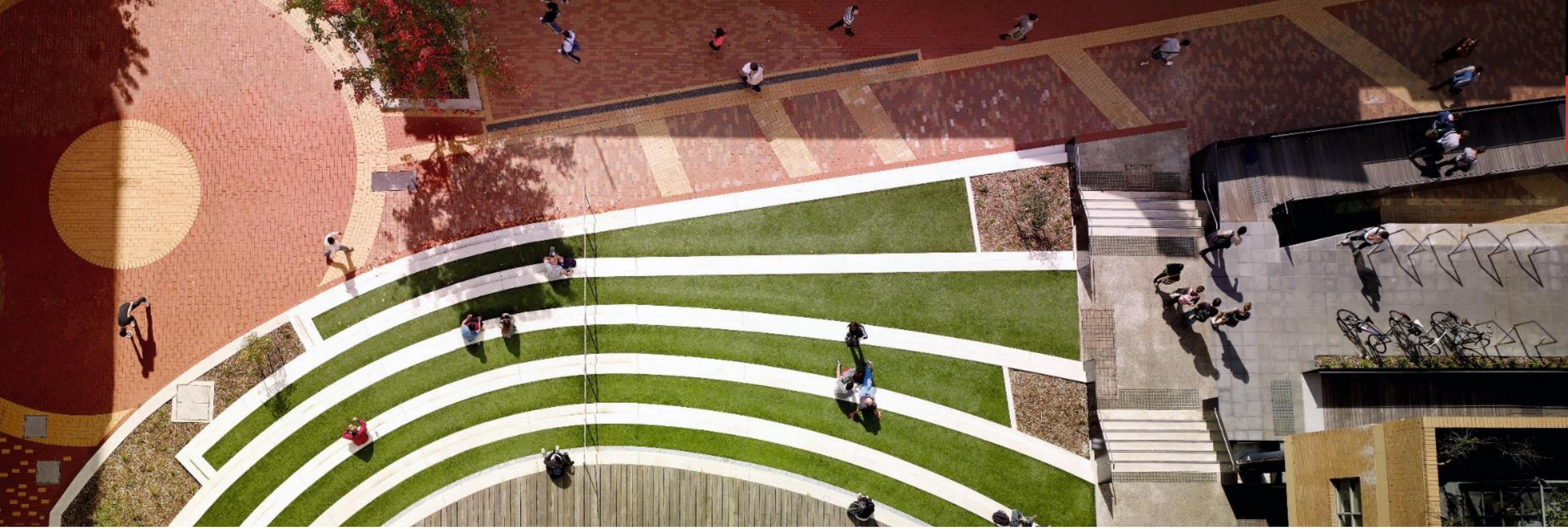


Introduction to Sequence Classification using KERAS

Dr. Matthew Smith, ADACS Senior Software Engineer

matthewrsmith@swin.edu.au

CRICOS 00111D
TOID 3059

WELCOME BACK

- After Lachlan's talk this morning, you (hopefully) have a clearer picture of the fundamentals of machine learning.
- The purpose of this session is to introduce you to Keras, and in a hands-on manner, apply this tool to solve a practical problem.

Please go ahead and log in (SSH in) to Ozstar now. Let us know if you run into troubles.

KNOW ING

HANDS-ON COMPONENT

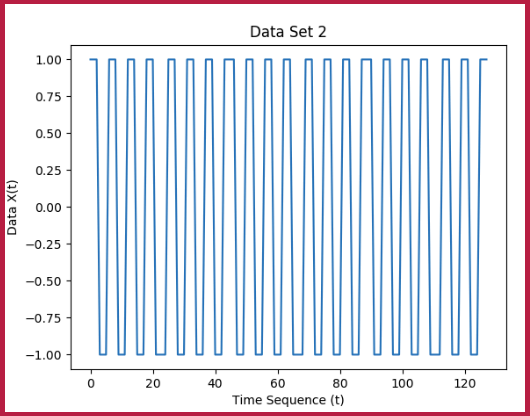
- Today's workshop is a hands-on workshop – as I'm talking, I am expecting you to be (i) logged in to Ozstar, (ii) ready to write code and execute them in a remote manner, and (iii) familiar with linux and the use of the Ozstar system in general.
- If you have not used Ozstar previously, this is a great chance - I'll take regular breaks during each session to make sure all is good.

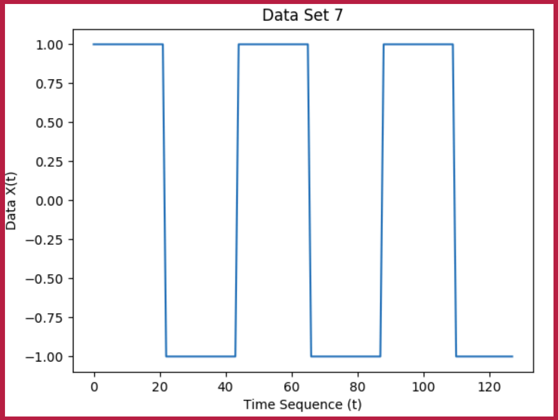
KNOW ING

PROBLEM DEFINITION

- Consider a binary classification problem and we are asking the AI to examine time series data from two different types of time series:

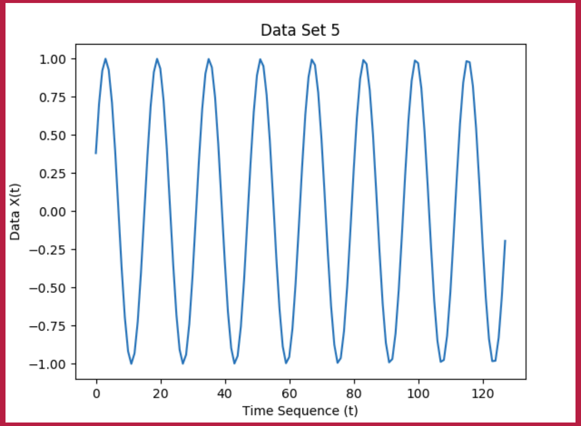
$$x(t) = square(0.1 + 0.3R_F)$$

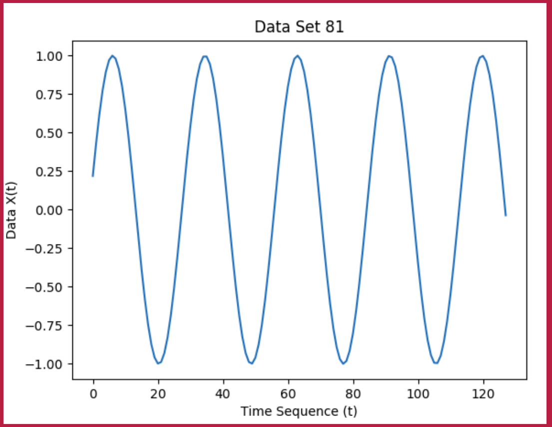




Group 0

$$x(t) = \sin(0.1 + 0.3R_F)$$

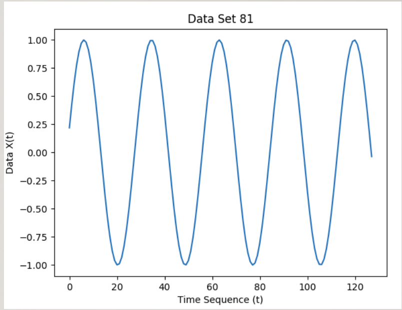


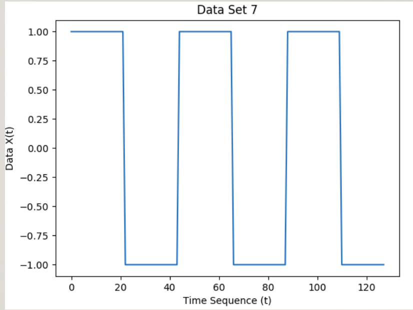


Group 1

PROBLEM DEFINITION

- Can we create a machine learning tool which is able to load the sequence of data from a file and be able to distinguish between a sine or square wave for an arbitrary frequency?
- This is the goal of this tutorial – to achieve this goal, we will use Keras - a high end API which runs on top of Tensorflow.
- To train our neural network, we will need to create training data sets.





PROBLEM DEFINITION

- You’ll need to make copies of these training sets in your home directories.
- The best thing to do is git clone them:
  - Load the git module:

```
module load git/2.16.0
```

  - Git clone the ADACS_ML_A repository:

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

  - Go into the Test and Train directories, and unzip the data files:

```
tar -xvf test.tar.gz
```

```
tar -xvf train.tar.gz
```

Go ahead and do this now. I’ll pause here until everyone has done it.

KNOW ING

PROBLEM DEFINITION

- The data for each time series – or sequence – is separated into two folders:
  - A training folder (./Train) which contains a large number of files used for training.
  - A testing folder (./Test) which contains the data we will use to test our model.
    In each directly, we see X files (containing time series) and Y files (containing the classification, 1 or 0).
    All files are binary, double precision.

X_159.dat	X_2.dat	X_6.dat	Y_10.dat	Y_140.dat	Y_181.dat
X_16.dat	X_20.dat	X_60.dat	Y_100.dat	Y_141.dat	Y_182.dat
X_160.dat	X_200.dat	X_61.dat	Y_101.dat	Y_142.dat	Y_183.dat
X_161.dat	X_21.dat	X_62.dat	Y_102.dat	Y_143.dat	Y_184.dat
X_162.dat	X_22.dat	X_63.dat	Y_103.dat	Y_144.dat	Y_185.dat
X_163.dat	X_23.dat	X_64.dat	Y_104.dat	Y_145.dat	Y_186.dat
X_164.dat	X_24.dat	X_65.dat	Y_105.dat	Y_146.dat	Y_187.dat
X_165.dat	X_25.dat	X_66.dat	Y_106.dat	Y_147.dat	Y_188.dat
X_166.dat	X_26.dat	X_67.dat	Y_107.dat	Y_148.dat	Y_189.dat
X_167.dat	X_27.dat	X_68.dat	Y_108.dat	Y_149.dat	Y_19.dat
X_168.dat	X_28.dat	X_69.dat	Y_109.dat	Y_15.dat	Y_190.dat
X_169.dat	X_29.dat	X_7.dat	Y_11.dat	Y_150.dat	Y_191.dat
X_17.dat	X_3.dat	X_70.dat	Y_110.dat	Y_151.dat	Y_192.dat

PROBLEM DEFINITION

- For your reference, these files were generated using the MATLAB functions included in the Train and Test folders.
- From the MATLAB command prompt, call Generate_Data(N) where N is an integer. This will create N data files, numbered from 1 to N.
- You can see how these data files are generated – approximately half of them will be sine waves, the other have square waves.
- We could modify these MATLAB scripts for multi-class classification problems easily.

```
function [u] = Generate_Data(N)
% Dr. Matthew Smith, Swinburne University of Technology
% Generate N data files, each containing a time series
% (i.e. sequence) corresponding to one of two classes:
% Y = 0 : Sine Wave
% Y = 1 : Square wave
% Each file will employ a different phase and frequency
% so give the RNN some degree of challenge.

Sequence_size = 128; % 128 values in each time series

for i = 1:1:N

    filename_x = sprintf('X_%d.dat', i);
    filename_y = sprintf('Y_%d.dat', i);
    if (rand() < 0.5)
        % Make a sin wave
        freq = 0.1 + rand()*0.3;
        y = 0;
        x = 1:1:128;
        fx = sin(freq*x);
    else
        % Make a square wave
        freq = 0.1 + rand()*0.3;
        y = 1;
        x = 1:1:128;
        fx = square(freq*x);
    end
    % Now to save each
    fileID = fopen(filename_x,'w');
    fwrite(fileID, fx, 'double');
    fclose(fileID);
    fileID = fopen(filename_y,'w');
```

PROBLEM DEFINITION

- The mission, theoretically, is pretty straight forward:
  - Using Python (Python 2.7 to be exact), load both the training sets and testing sets of data.
  - Use Keras / Tensorflow to build a Recurring Neural Network (RNN) with numerous layers to create a model.
  - Use this model with our testing data to check its accuracy.
  - We will use python's matplotlib to perform some visualization of the accuracy obtained during the training process.

PROBLEM DEFINITION

- Questions we want to investigate at this stage are:
  - How do we use the Ozstar environment to perform this work?
  - How does the number of training data sets used influence the convergence and final accuracy?
  - How many epochs are required to see acceptable results?

PREPARATION – OZSTAR MODULES

- When using Ozstar to perform computation, only several very basic tools are loaded when you log in.
- To load functionality into our Ozstar environment, we use modules to load what we need.

```
module purge all
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 pandas/0.22.0-python-2.7.14
module load keras/2.1.4-python-2.7.14
```

- The modules we require are shown on the right – we could load them in one-by-one, but that would be a waste of time.
- Load the modules by typing ”. script.sh <enter>” (no quotation marks).

You’ll notice script.sh was included with your git clone. Load it now – I’ll check.

KNOW ING

INTRODUCTION TO KERAS

- Today's tutorial includes several python files:
  - train.py – the main script which, when called, loads the training and test data sets, creates the Keras model and defines the neural network, performs the training and tests the model.
  - utilities.py – a script containing simple functions for loading data from files and plotting using matplotlib. This is not called directly; it contains functions called by train.py and view.py,
  - view.py – a stand-alone script which is used to inspect training data for your own verification purposes (i.e. sanity checking).

REVIEW OF TRAIN.PY

TRAIN.PY

- I'd like you to follow along as I browse through train.py.
- Move into the directory where your files are kept, and open the file with a text editor.
- If you are unfamiliar with editing codes through SSH, try nano:

  nano train.py <enter>

```
# train.py
# Written by Dr. Matthew Smith, Swinburne University of Technology
# Prepared as training material for ADACS Machine Learning workshop
# This is an example of time series (sequence) classification
# for a binary classification problen.

# Import modules
import numpy as np
from keras.models import Sequential
from keras.layers import Dense
#from keras.layers import LSTM
from keras.layers import Activation
from keras.utils import plot_model
from utilities import *

# Create training arrays
# In this demonstration I create our numpy arrays and then
# load each time sequence in one-by-one.
N_train = 200        # Number of elements to train
N_sequence = 128     # Length of each piece of data
N_epochs = 300       # Num

# Create the training seq
X_train = np.empty([N_tra
Y_train = np.empty(N_trai
```

Nano does support syntax highlighting*

* Not a paid advertisement

TRAIN.PY

- As with most python scripts, we start by loading modules.
- After loading modules, we define the number of training data sets to load (N_train) – here, we have 200 data sets.
- Each data set contains a time series with 128 elements (N_sequence).

```
# train.py
# Written by Dr. Matthew Smith, Swinburne University of Technology
# Prepared as training material for ADACS Machine Learning workshop
# This is an example of time series (sequence) classification
# for a binary classification problen.

# Import modules
import numpy as np
from keras.models import Sequential
from keras.layers import Dense
#from keras.layers import LSTM
from keras.layers import Activation
from keras.utils import plot_model
from utilities import *

# Create training arrays
# In this demonstration I create our numpy arrays and then
# load each time sequence in one-by-one.
N_train = 200          # Number of elements to train
N_sequence = 128       # Length of each piece of data
N_epochs = 300         # Number of epochs

# Create the training sequence data (X) and each set's classificat
X_train = np.empty([N_train, N_sequence])
Y_train = np.empty(N_train)
```

KNOW ING

TRAIN.PY

- We only load in the parts of Keras which we need.
- In this work, we are performing a neural network analysis on time series data - in keras, this form of analysis is known as a Sequential analysis – hence, we need to import Sequential.

```
# train.py
# Written by Dr. Matthew Smith, Swinburne University of Technology
# Prepared as training material for ADACS Machine Learning workshop
# This is an example of time series (sequence) classification
# for a binary classification problen.

# Import modules
import numpy as np
from keras.models import Sequential
from keras.layers import Dense
#from keras.layers import LSTM
from keras.layers import Activation
from keras.utils import plot_model
from utilities import *

# Create training arrays
# In this demonstration I create our numpy arrays and then
# load each time sequence in one-by-one.
N_train = 200           # Number of elements to train
N_sequence = 128        # Length of each piece of data
N_epochs = 300          # Number of epochs

# Create the training sequence data (X) and each set's classificat
X_train = np.empty([N_train, N_sequence])
Y_train = np.empty(N_train)
```

KNOW ING

TRAIN.PY - SEQUENTIAL

- To employ Neural Networks for learning over time sequences of data, we use a Recurring Neural Network (RNN).
- A Recurrent neural networks is a deep neural neural network which has, as the name suggests, recurring inputs to the hidden layer i.e. the output from a hidden layer is fed back to itself.

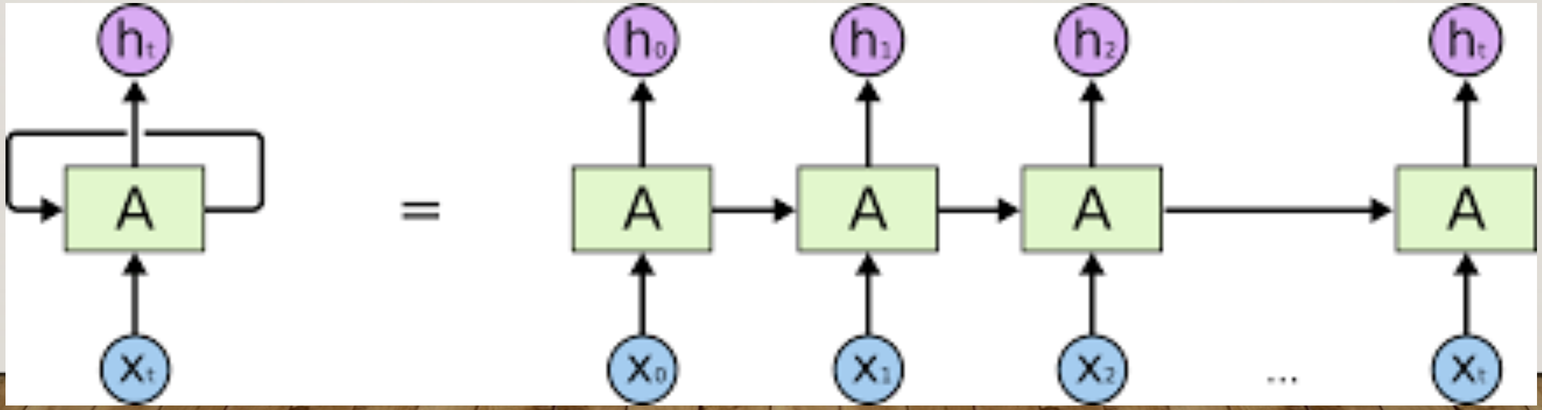


Image: https://medium.com/themlblog/time-series-analysis-using-recurrent-neural-networks-in-tensorflow-2a0478b00be7

TRAIN.PY - SEQUENTIAL

- Here, A – our neural network – may contain numerous layers - is repeatedly fed consecutive data from our time series. This is to ensure that the history of our time data is taken into account – that we have what we might describe as a Neural Memory.
- Neural memory is the ability imparted to a model to retain the input from previous time steps when the input is sequential.
- The same approach is used to treat image classification – but this falls outside the scope of this workshop.

TRAIN.PY - SEQUENTIAL

- One potential problem with very large data sets is that information – which might tend to be very important on a small time scale in the the large time sequence – tends to disappear into the background when the process is repeated over very large time steps.
- We can use an approach called Long-short Term Memory networks(LSTM) to solve this problem.
- In this case, we won't – our sequences are quite short, and periodic – but modification of this script to perform this improvement over conventional RNN is quite simple.

KNOW ING

TRAIN.PY

- Since – in this case – our data is small, we can load it all at once into memory.
- We create two numpy arrays (X_train and Y_train) to hold our training data – initially empty.
- We then load each file (one by one) using the read_training_data function contained in utilities.py
- We repeat the process for the testing data set.

```python
# Create the training sequence data (X) and each set's classification (Y).
X_train = np.empty([N_train, N_sequence])
Y_train = np.empty(N_train)

# Load the data from file
for x in range(N_train):
        # This will create x = 0, 1, 2...to N_train-1
        X_train[x,], Y_train[x] = read_training_data(x+1, N_sequence)

# Also create the numpy arrays for the testing data set
N_test = 50
X_test = np.empty([N_test, N_sequence])
Y_test = np.empty(N_test)
for x in range(N_test):
        X_test[x,], Y_test[x] = read_test_data(x+1, N_sequence)
```

We might have used only one variable (X_train) for both testing and training using splitting in Keras – you can google this if you are interested.

Keras has numerous strategies for managing data which is too large to fit into memory in a single instance, using data generators.

TRAIN.PY

- We start by creating a Keras Sequential model:

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

# Configure our RNN by adding neural layers with activation functions
model.add(Dense(16, activation='relu',input_dim=N_sequence))
model.add(Dense(8, activation='tanh'))
model.add(Dense(1, activation='sigmoid'))
```

- The model variable holds our neural network, weights and all parameters.
- The Sequential class also has a large number of class functions, some of which we will see later on in this tutorial.

KNOW ING

TRAIN.PY

- We add hidden Neural layers to the model using the add() function.

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

# Configure our RNN by adding neural layers with activation functions
model.add(Dense(16, activation='relu',input_dim=N_sequence))
model.add(Dense(8, activation='tanh'))
model.add(Dense(1, activation='sigmoid'))
```

- The first layer we are adding is a densely connected neural layer with an input of N_sequence – we are inputting each piece of time series data as an input - and an intermediate output of 16 neurons.
- Each layer has an associated activation function – in this case, it is 'relu' - Rectified Linear Unit.

KNOW ING

TRAIN.PY

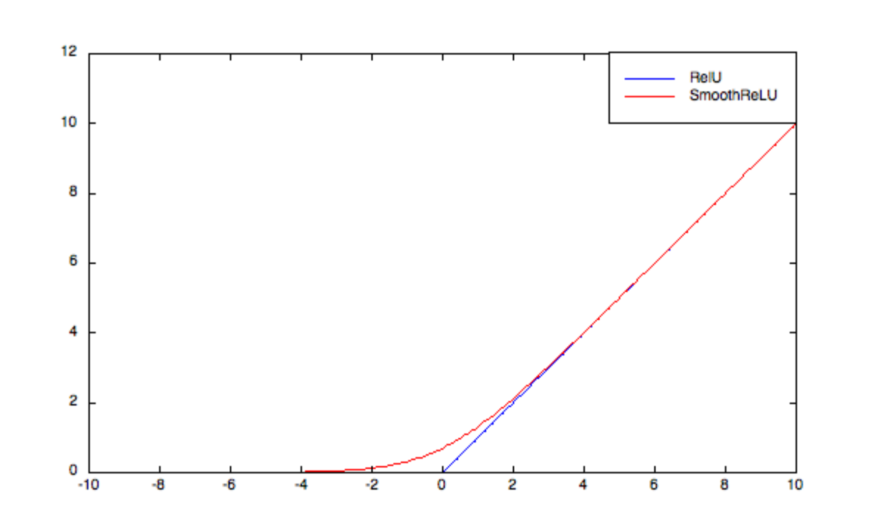
- The relu function is defined as the maximum positive part of its argument tensor:

$$f(x) = \max(0, x)$$

- It has found popular use in deep learning networks in recent years, but is discontinuous. A smooth option is SmoothReLU:

$$f(x) = \log(1 + \exp(x))$$

- We use relu due to its ability to pass gradient information between subsequent iterations – allowing us to avoid the use of LSTM for now.



TRAIN.PY

- We then add another layer, using a different activation function (which you might comment out)

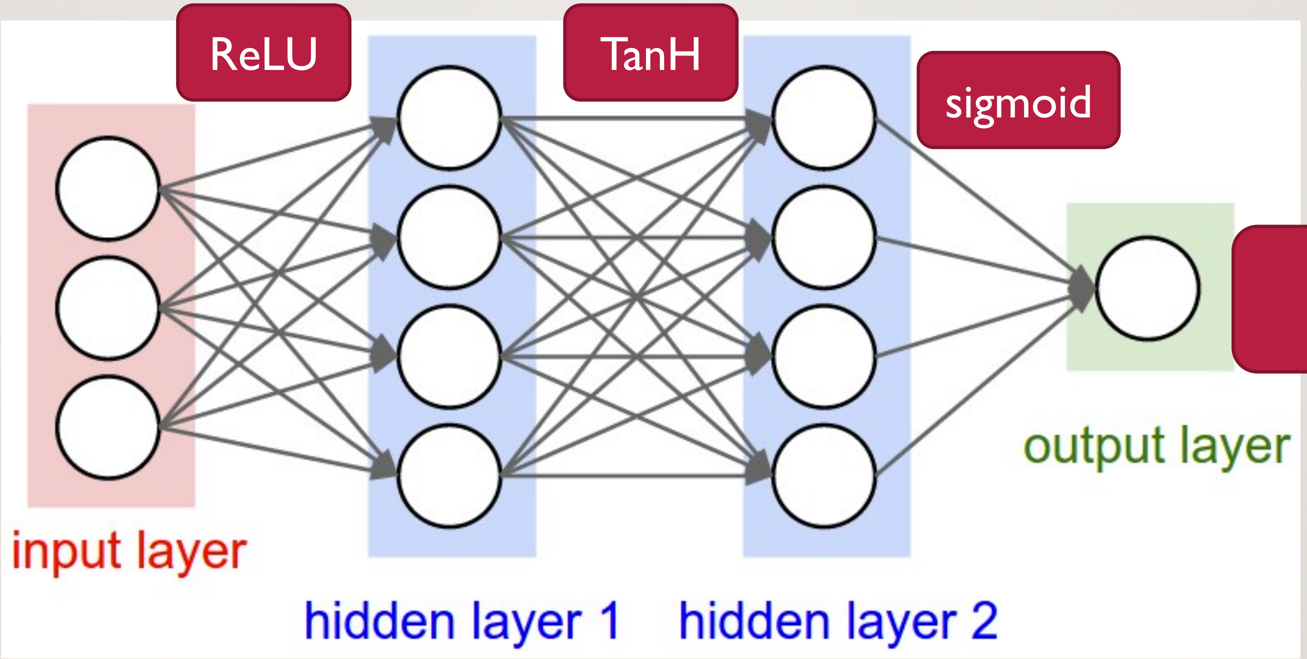
```
# Create our Keras model - an RNN (in Keras this is a Sequence)
model = Sequential()

# Configure our RNN by adding neural layers with activation functions
model.add(Dense(16, activation='relu',input_dim=N_sequence))
model.add(Dense(8, activation='tanh'))
model.add(Dense(1, activation='sigmoid'))
```

- This time we use the tanh function – this is a non-linear function, which allows us to introduce non-linear dependences into our neural network.
- We've also changed the number of neurons in the layer to 8 – all of which are fully connected (dense).

TRAIN.PY

- The result is something like this – only don't pay attention to the number of neurons in each layer.



Input layer = our time series data (128 neurons)

1 neuron

16 neurons

8 neurons

KNOW ING

TRAIN.PY

- We need to compile our Keras model before we start:

```
# Compile model and print summary
model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['accuracy'])
print(model.summary())

# 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)
```

- An optimizer is a function designed to increase learning speed – we can specify these separately if we wish to alter the learning rate etc. – find more info here: https://keras.io/optimizers/
- Our loss function is the function used to measure the effectiveness of the learning (for the optimizer) – the binary cross entropy function has found favour recently for binary classification.

KNOW ING

TRAIN.PY

- Finally we can perform our fit:

```
# Compile model and print summary
model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['accuracy'])
print(model.summary())

# 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)
```

- The training process is repeated for all training sets N_epochs time – this value should be large enough that we demonstrate convergence on the accuracy computed during training.
- To inspect this, we plot the history using the plot_history function inside utilities.py

TRAIN.PY

- Note:

```
# Compile model and print summary
model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['accuracy'])
print(model.summary())

# 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)
```

- To be able to see this plot, you may need to jump through a few hoops:
  - You will need to ensure X11 forwarding is enabled (add –X to ssh login)
  - On MAC you may need XQuartz installed and running.

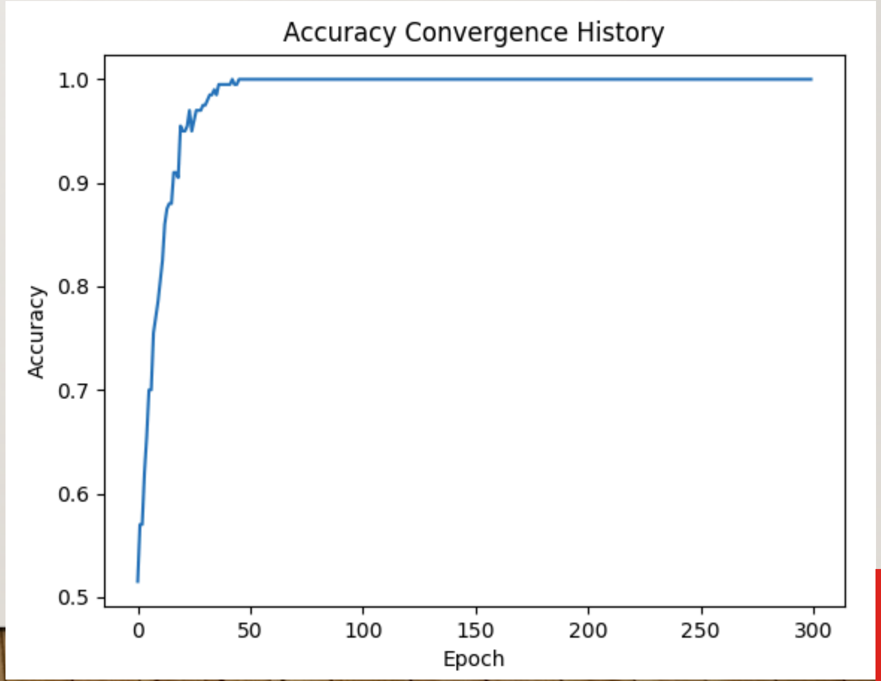
RESULTS

- To run your script on the head node (tsk tsk), move to the directory where your python scripts are and type:

python train.py <enter>

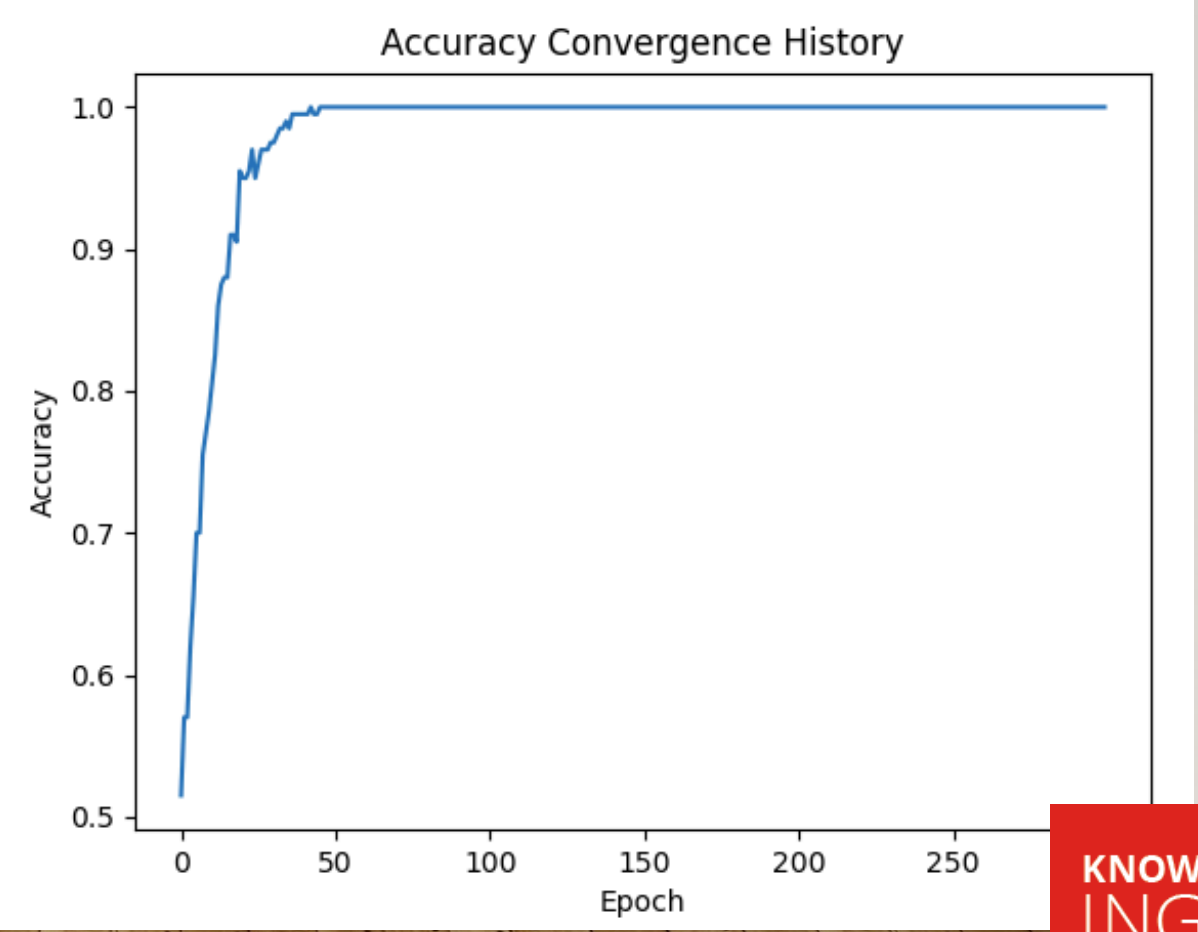
Run this now. If you are forwarding X (i.e. enabling graphics over SSH), you ought to see a graph showing training accuracy.

```
Epoch 294/300
200/200 [==============================] - 0s 16us/step - loss: 1.3949e-07 - acc: 1.0000
Epoch 295/300
200/200 [==============================] - 0s 16us/step - loss: 1.3738e-07 - acc: 1.0000
Epoch 296/300
200/200 [==============================] - 0s 16us/step - loss: 1.3538e-07 - acc: 1.0000
Epoch 297/300
200/200 [==============================] - 0s 16us/step - loss: 1.3339e-07 - acc: 1.0000
Epoch 298/300
200/200 [==============================] - 0s 16us/step - loss: 1.3156e-07 - acc: 1.0000
Epoch 299/300
200/200 [==============================] - 0s 16us/step - loss: 1.2975e-07 - acc: 1.0000
Epoch 300/300
200/200 [==============================] - 0s 16us/step - loss: 1.2801e-07 - acc: 1.0000
Evaluating Test Set
50/50 [==============================] - 0s 221us/step
Accuracy: 100.00%
```



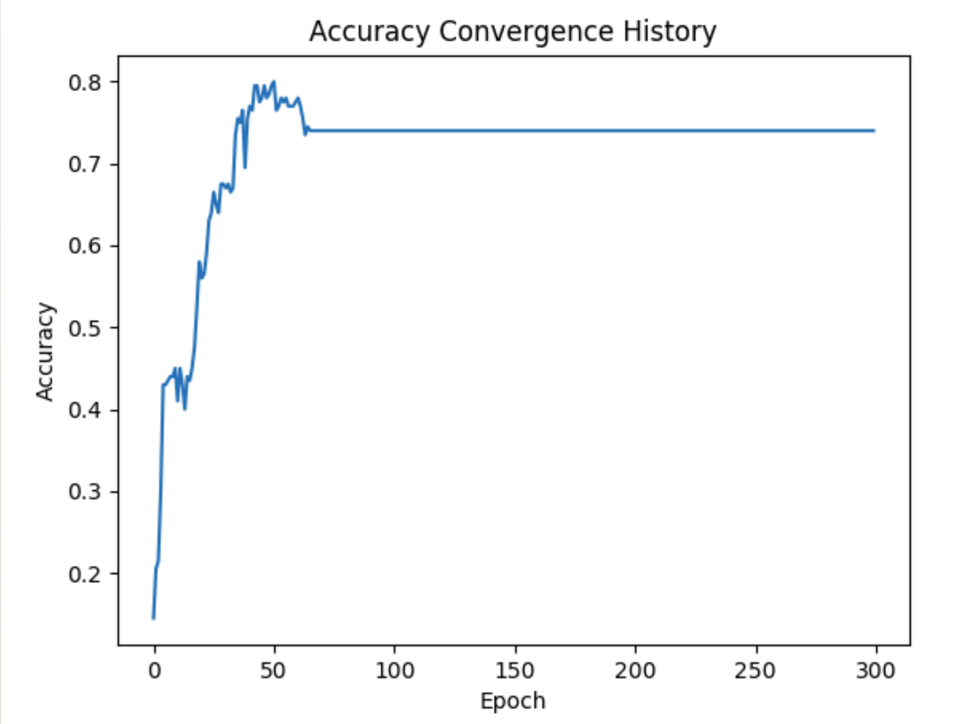
RESULTS

- After running the script with N_epochs = 300 with 200 training sets, you should see convergence look like this.
- This was with 2 layers of neurons – you should experiment by adding various numbers (and sizes) of neuron layers; it will influence the accuracy convergence.



DISCUSSION - ACTIVATORS

- Consider the case where we have a single hidden neural layer and no activation functions.
- Hence, the relationship between input and output is strictly linear.
- We can see that the machine is incapable of learning – indicating to us that some manner of non-linearity exists.



https://towardsdatascience.com/exploring-activation-functions-for-neural-networks-73498da59b02

ACTIVITY TIME

- In the time remaining:
  - Experiment with the activation functions and their influence on training accuracy convergence vs epoch.
  - Try adding (or removing) additional hidden layers of neurons with different numbers of neurons.
  - Instead of running the job on the head node, submit the job properly (hopefully Ozstar is not still busy; calculations are light.)

If you are happy with your results, head off to lunch. We will resume at 2pm.

KNOW ING