INTERNATIONAL JOURNAL OF ADVANCE RESEARCH, IDEAS AND INNOVATIONS IN TECHNOLOGY

ISSN: 2454-132X
Impact factor: 6.078
(Volume 6, Issue 3)
Available online at: www.ijariit.com

Classify radio signals from space

Tarit Sengupta
info.taritsengupta01@gmail.com
Techno Main Salt Lake, Kolkata, West Bengal

ABSTRACT

The data we are going to use consists of 2D spectrograms of deep space radio signals collected by the Allen Telescope Array at the SETI Institute. We will treat the spectrograms as images to train an image classification model to classify the signals into one of four classes. By the end of the project, you will have built and trained a convolutional neural network from scratch using Keras to classify signals from space.

Keywords— *Radio Signals, Space Signals, Seti Data, Cnn Model, Model Train*

1. INTRODUCTION

In this project we use 2D Spectrograms of deep-space radio signals collected by the antennas at the SETI (The search for extra-terrestrial intelligence**)** institute.

What they do is they use their Allen telescope array and wide range of antennas to scan the night sky for very faint radio signals, from the outer space. What we will do is we'll treat these 2-D Spectrograms as images and put them in our image classifier to classify these faint radio signals into one of the four categories.

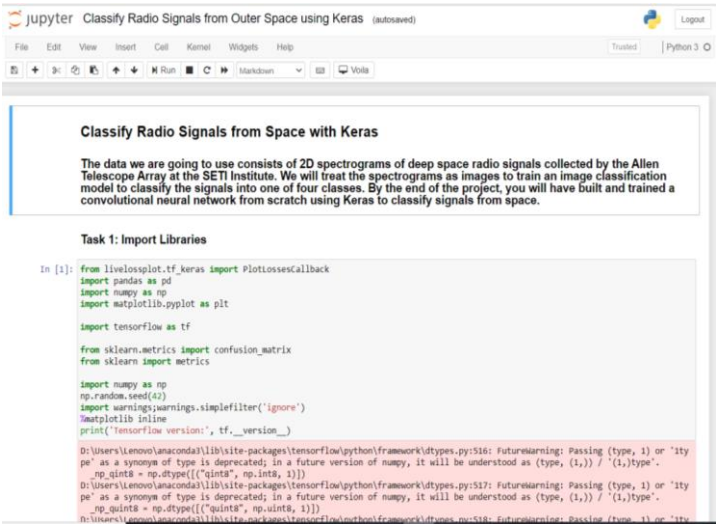
To give you a bit more context, at the SETI institute they use the Allen telescope array situated in northern California to scan the sky at various radio frequencies to observe the star system with known exoplanets.
The goal is to search for faint but persistent signals.

The current signal detection system is programmed for only particular kinds of signals such as narrow-band carrier waves however, the detection system sometimes triggers the signals that are not narrow-band signals with some unknown efficiency and are also not explicitly known frequency interference. So there seem to be various categories of these kinds of events that have been observed in the recent past so our goal for the next few minutes is to build an image classification model to classify these signals accurately in real-time so this may allow the signal detection system to make better observational decisions and thereby increasing the efficiency of the night scans, allowing for explicit detection of signal types.

So the original signals were not 2-D spectrograms but were time-series data collected and downloaded by Seti. We're going to work with 2-D spectrograms which were created by transforming the input time-series data. So, we're going to use the spectrograms as images to train or classification model. By the end of this blog, you will learn to build & train a CNN by scratch to classify these radio signals from space.

I'm expecting that you have a prior knowledge about coding in Python and have a basic idea about how a neural network performs under the hood especially convolutional neural network as I won't be getting in the maths. We'll begin with importing the libraries.

We'll be using Tensorflow v 2.2.0.



2. LOAD AND PRE-PROCESS SETI DATA

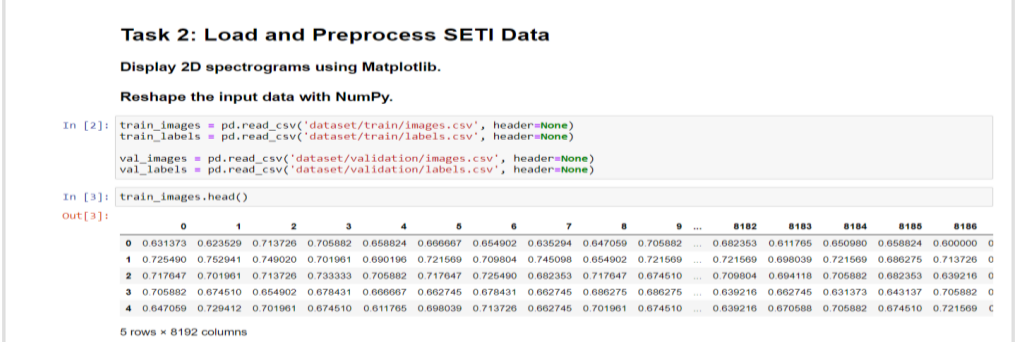
We use pandas to read the CSV file where the images are stored. But how can images be stored in a CSV file?

The spectrograph images were converted into their raw pixel intensity values and were normalized so the values lie between 0 and 1. They are then converted into an array by stretching them. Therefore, each row of the CSV file corresponds to a single image.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	0.631373	0.623529	0.713726	0.705882	0.658824	0.666667	0.654902	0.635294	0.647059	0.705882	0.729412	0.72549	0.619608	0.67451
2	0.72549	0.752941	0.74902	0.701961	0.690196	0.721569	0.709804	0.745098	0.654902	0.721569	0.678431	0.709804	0.713726	0.686275
3	0.717647	0.701961	0.713726	0.733333	0.705882	0.717647	0.72549	0.682353	0.717647	0.67451	0.690196	0.670588	0.662745	0.666667
4	0.705882	0.67451	0.654902	0.678431	0.666667	0.662745	0.678431	0.662745	0.686275	0.686275	0.631373	0.65098	0.670588	0.737255
5	0.647059	0.729412	0.701961	0.67451	0.611765	0.698039	0.713726	0.662745	0.701961	0.67451	0.631373	0.709804	0.694118	0.698039
6	0.694118	0.682353	0.705882	0.705882	0.666667	0.694118	0.67451	0.713726	0.690196	0.709804	0.756863	0.690196	0.67451	0.721569
7	0.717647	0.686275	0.760784	0.741176	0.709804	0.72549	0.733333	0.698039	0.654902	0.721569	0.72549	0.729412	0.721569	0.717647
8	0.713726	0.713726	0.658824	0.690196	0.682353	0.705882	0.709804	0.717647	0.733333	0.733333	0.701961	0.72549	0.72549	0.705882
9	0.658824	0.678431	0.729412	0.690196	0.705882	0.678431	0.658824	0.670588	0.713726	0.670588	0.654902	0.682353	0.647059	0.654902
10	0.721569	0.729412	0.764706	0.709804	0.701961	0.658824	0.709804	0.709804	0.698039	0.717647	0.694118	0.72549	0.694118	0.686275
11	0.635294	0.647059	0.686275	0.705882	0.670588	0.666667	0.666667	0.670588	0.65098	0.705882	0.72549	0.686275	0.72549	0.647059
12	0.709804	0.713726	0.729412	0.698039	0.647059	0.752941	0.721569	0.694118	0.690196	0.666667	0.694118	0.678431	0.694118	0.709804
13	0.686275	0.67451	0.717647	0.709804	0.713726	0.709804	0.698039	0.694118	0.721569	0.72549	0.709804	0.72549	0.74902	0.733333
14	0.666667	0.662745	0.67451	0.690196	0.647059	0.690196	0.764706	0.666667	0.682353	0.701961	0.682353	0.67451	0.694118	0.705882
15	0.694118	0.737255	0.705882	0.701961	0.709804	0.721569	0.709804	0.67451	0.745098	0.67451	0.682353	0.72549	0.760784	0.709804
16	0.713726	0.647059	0.65098	0.698039	0.654902	0.694118	0.717647	0.67451	0.670588	0.670588	0.686275	0.694118	0.678431	0.65098
17	0.756863	0.737255	0.701961	0.658824	0.717647	0.701961	0.729412	0.741176	0.698039	0.694118	0.705882	0.701961	0.701961	0.713726
18	0.729412	0.721569	0.701961	0.67451	0.662745	0.67451	0.678431	0.72549	0.733333	0.694118	0.682353	0.701961	0.709804	0.678431

The label was found to be one hot encoded in to a vector of 1,4(no. of classes).

- 1,0,0,0 is squiggle
- 0,1,0,0 is Narrow-band signal
- 0,0,1,0 is Noise
- 0,0,0,1 is Narrow-band-drd signal



We read the files into our pandas DataFrame and specify that we don't require a header as we have no header in the CSV file. We then visualize the DataFrames using the .head() method.

and see the following output:

```
In [3]: train_images.head()
```

Out[3]:

	0	1	2	3	4	5	6	7	8	9	...	8182	8183	8184	8185	8186	
0	0.631373	0.623529	0.713726	0.705882	0.658824	0.666667	0.654902	0.635294	0.647059	0.705882	...	0.682353	0.611765	0.650980	0.658824	0.600000	0
1	0.725490	0.752941	0.749020	0.701961	0.690196	0.721569	0.709804	0.745098	0.654902	0.721569	...	0.721569	0.698039	0.721569	0.686275	0.713726	0
2	0.717647	0.701961	0.713726	0.733333	0.705882	0.717647	0.725490	0.682353	0.717647	0.674510	...	0.709804	0.694118	0.705882	0.682353	0.639216	0
3	0.705882	0.674510	0.654902	0.678431	0.666667	0.662745	0.678431	0.662745	0.686275	0.686275	...	0.639216	0.662745	0.631373	0.643137	0.705882	0
4	0.647059	0.729412	0.701961	0.674510	0.611765	0.698039	0.713726	0.662745	0.701961	0.674510	...	0.639216	0.670588	0.705882	0.674510	0.721569	0

5 rows × 8192 columns

```
In [4]: train_labels.head()
```

Out[4]:

	0	1	2	3
0	1.0	0.0	0.0	0.0
1	1.0	0.0	0.0	0.0
2	1.0	0.0	0.0	0.0
3	1.0	0.0	0.0	0.0
4	1.0	0.0	0.0	0.0

We then try to see the shape of the DataFrames and we find that the output:

```
In [5]: print("Training set shape:", train_images.shape, train_labels.shape)
        print("Validation set shape:", val_images.shape, val_labels.shape)

        Training set shape: (3200, 8192) (3200, 4)
        Validation set shape: (800, 8192) (800, 4)
```

8192 is nothing but 64*128, therefore we'll reshape them into 64*128(representing the width and height of the image).

```
In [6]: x_train = train_images.values.reshape(3200, 64, 128, 1)
        x_val = val_images.values.reshape(800, 64, 128, 1)

        y_train = train_labels.values
        y_val = val_labels.values
```

Since all of the images were converted into 2-D spectrograms we didn't have information about RGB channels therefore we put the 1 in the last dimension or else we would have put a 3 had it been coloured or colour had an importance here. Also, we converted the data frames into the array format(.values) because our neural network can't take in the Data Frame format and our image classifier wants the data to be in that specific format which was achieved after reshaping.

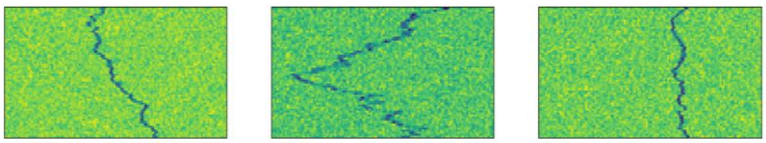
Let's try to visualize the images using matplotlib:

Task 3: Plot 2D Spectrograms

Generate batches of tensor image data with real-time data augmentation.

Specify paths to training and validation image directories and generates batches of augmented data.

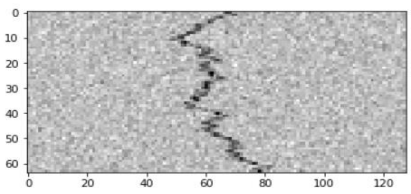
```
In [25]: plt.figure(0, figsize=(16,16))
         for i in range(1,4):
             plt.subplot(1,3,i)
             img = np.squeeze(x_train[np.random.randint(0, x_train.shape[1])])
             plt.xticks([])
             plt.yticks([])
             plt.imshow(img)
```



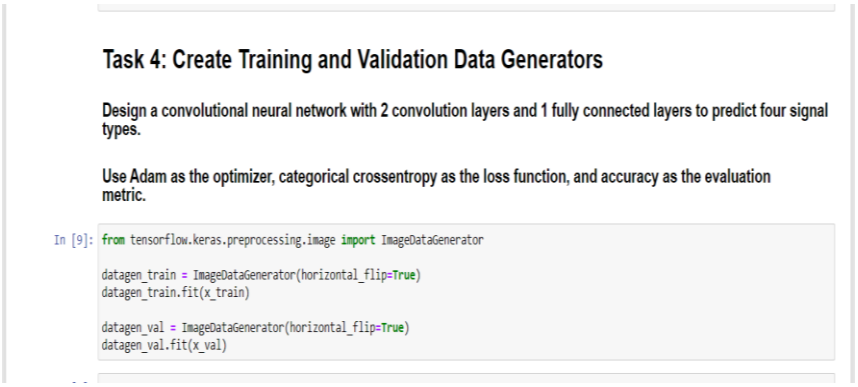
Looks like all three of the randomly selected images are of the type Narrow-band.

You can re-run the function to see a different set of images. The real benefit of deep learning comes when we train on batches of data, so let's generate batches of tensor image data with real time data augmentation. Feel free to experiment with your techniques of augmenting your data.. For now I'll just be flipping them horizontally.

```
In [23]: plt.imshow(np.squeeze(x_train[3]), cmap="gray");
```



3. CREATE TRAINING AND VALIDATION DATA GENERATORS



Task 4: Create Training and Validation Data Generators

Design a convolutional neural network with 2 convolution layers and 1 fully connected layers to predict four signal types.

Use Adam as the optimizer, categorical crossentropy as the loss function, and accuracy as the evaluation metric.

In [9]:
```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

datagen_train = ImageDataGenerator(horizontal_flip=True)
datagen_train.fit(x_train)

datagen_val = ImageDataGenerator(horizontal_flip=True)
datagen_val.fit(x_val)
```

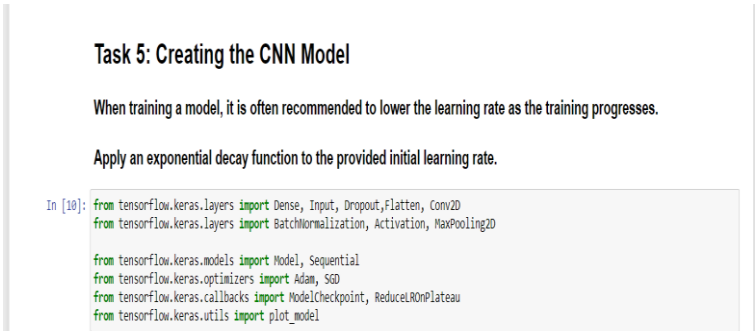
As the data was already processed and normalized, we don't need to do a lot of pre-processing. We are finally ready to build our model.

Let me give you a brief introduction to CNN if you're a bit rusty of the knowledge on that. So, CNNs are just a type of feed-forward neural network consisting of multiple layers of neurons that have learnable weights and biases, so each neuron in a layer receives an input from a proceeding layer processes the input and optionally follows it with non-linearity. So that your model not just learns a linear sum of the data but also complex non-linear functions.

The network for CNN has multiple layers such as convolution layers, maxpool layers for down-sampling, dropout for regularization, and all these layers followed by one or more fully-connected layers at the end. So, at each layer, a small neuron process portion of input images and output of these collections are then tiled so that the input region overlap to obtain a high-resolution representation of the input image, and this process is repeated for every such layer.

The bottom line is that CNN takes a complex pattern in images and breaks them down to simple patterns through multiple hierarchical layers. Max-pooling is simply a non-linear down sampler, it partitions the input image into a set of rectangles and then finds the max value of that region.

We will now use Keras to implement our CNN. We'll first begin by importing all the util files from Keras and tf:



Task 5: Creating the CNN Model

When training a model, it is often recommended to lower the learning rate as the training progresses.

Apply an exponential decay function to the provided initial learning rate.

In [10]:
```
from tensorflow.keras.layers import Dense, Input, Dropout,Flatten, Conv2D
from tensorflow.keras.layers import BatchNormalization, Activation, MaxPooling2D

from tensorflow.keras.models import Model, Sequential
from tensorflow.keras.optimizers import Adam, SGD
from tensorflow.keras.callbacks import ModelCheckpoint, ReduceLROnPlateau
from tensorflow.keras.utils import plot_model
```

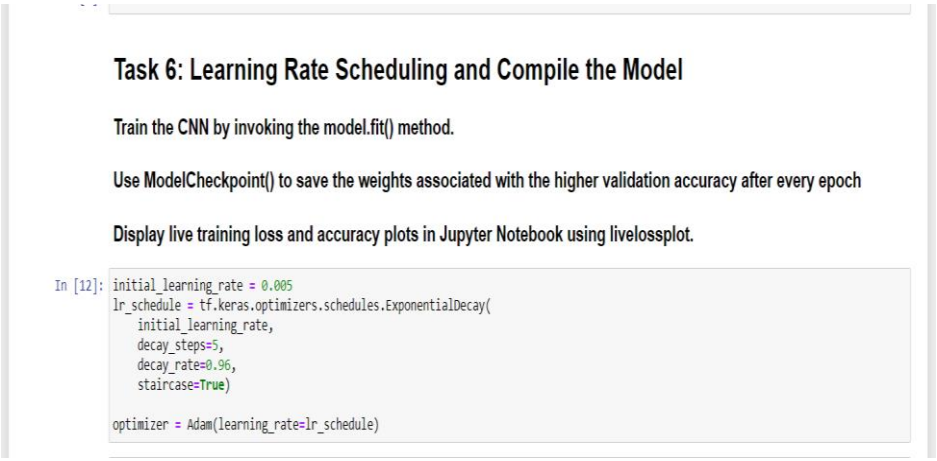
Here's the implementation of the model in Keras:

In [11]:
```
# Initialising the CNN
model = Sequential()

# 1st Convolution
model.add(Conv2D(32,(5,5), padding='same', input_shape=(64, 128,1)))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

# 2nd Convolution layer
model.add(Conv2D(64,(5,5), padding='same'))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

# Flattening
model.add(Flatten())

# Fully connected layer
model.add(Dense(1024))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(Dropout(0.4))

model.add(Dense(4, activation='softmax'))
```

```
WARNING:tensorflow:From D:\Users\Lenovo\anaconda3\lib\site-packages\tensorflow\python\ops\init_ops.py:1251: calling VarianceSca
ling.__init__ (from tensorflow.python.ops.init_ops) with dtype is deprecated and will be removed in a future version.
Instructions for updating:
Call initializer instance with the dtype argument instead of passing it to the constructor
```


Before we compile the model let's define a learning rate scheduler. Use of a scheduler is to decay the learning rate after some 'time'.

4. LEARNING RATE SCHEDULING AND COMPILE THE MODEL



Task 6: Learning Rate Scheduling and Compile the Model

Train the CNN by invoking the model.fit() method.

Use ModelCheckpoint() to save the weights associated with the higher validation accuracy after every epoch

Display live training loss and accuracy plots in Jupyter Notebook using livelossplot.

```
In [12]: initial_learning_rate = 0.005
         lr_schedule = tf.keras.optimizers.schedules.ExponentialDecay(
             initial_learning_rate,
             decay_steps=5,
             decay_rate=0.96,
             staircase=True)

         optimizer = Adam(learning_rate=lr_schedule)
```

This will keep mitigating the learning rate like 0.005*(0.96**5) = 0.004076 after each of the specified step. Here's the summary of the model:

```
In [13]: model.compile(optimizer=optimizer, loss='categorical_crossentropy', metrics=['accuracy'])
         model.summary()

Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 64, 128, 32)       832
_________________________________________________________________
batch_normalization (BatchNo (None, 64, 128, 32)       128
_________________________________________________________________
activation (Activation)      (None, 64, 128, 32)       0
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 32, 64, 32)        0
_________________________________________________________________
dropout (Dropout)            (None, 32, 64, 32)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 32, 64, 64)        51264
_________________________________________________________________
batch_normalization_1 (Batch (None, 32, 64, 64)        256
_________________________________________________________________
activation_1 (Activation)    (None, 32, 64, 64)        0
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 16, 32, 64)        0
_________________________________________________________________
dropout_1 (Dropout)          (None, 16, 32, 64)        0
_________________________________________________________________
flatten (Flatten)            (None, 32768)             0
_________________________________________________________________
dense (Dense)                (None, 1024)              33555456
_________________________________________________________________
batch_normalization_2 (Batch (None, 1024)              4096
_________________________________________________________________
activation_2 (Activation)    (None, 1024)              0
_________________________________________________________________
dropout_2 (Dropout)          (None, 1024)              0
_________________________________________________________________
dense_1 (Dense)              (None, 4)                 4100
=================================================================
Total params: 33,616,132
Trainable params: 33,613,892
Non-trainable params: 2,240
_________________________________________________________________
```

5. TRAIN THE MODEL

Before we start training our model, we need to define some call-backs if we are interested in saving our model at certain checkpoints or on certain optimizations to have the least validation loss.

Task 7: Training the Model

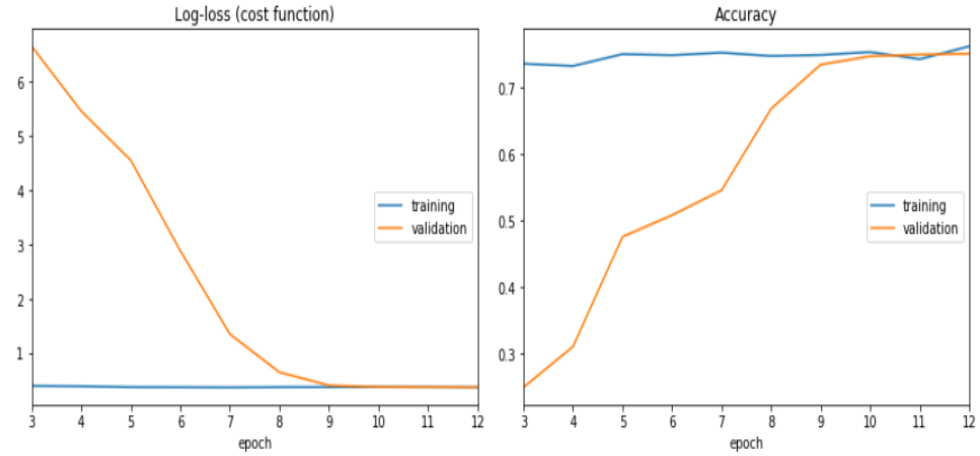
Evaluate the CNN by invoking the model.fit() method.

Obtain the classification report to note the precision and recall of your classifier.

In [14]:
```
checkpoint = ModelCheckpoint("model_weights.h5", monitor='val_loss',
                             save_weights_only=True, mode='min', verbose=0)
callbacks = [PlotLossesCallback(), checkpoint]#, reduce_lr]
batch_size = 32
history = model.fit(
    datagen_train.flow(x_train, y_train, batch_size=batch_size, shuffle=True),
    steps_per_epoch=len(x_train)//batch_size,
    validation_data = datagen_val.flow(x_val, y_val, batch_size=batch_size, shuffle=True),
    validation_steps = len(x_val)//batch_size,
    epochs=12,
    callbacks=callbacks
)
```

History object is the place we need to pass in our data generator that we created in the previous step.
The model train for about 10 mins on Google Colab without GPU or TPU.

```
Epoch 1/12
100/100 [==============================] - 115s 1s/step - loss: 0.5994 - accuracy: 0.6978 - val_loss: 3.7288 - val_accuracy: 0.2500
Epoch 2/12
100/100 [==============================] - 114s 1s/step - loss: 0.3997 - accuracy: 0.7472 - val_loss: 5.7785 - val_accuracy: 0.2612
Epoch 3/12
100/100 [==============================] - 116s 1s/step - loss: 0.3833 - accuracy: 0.7506 - val_loss: 5.9556 - val_accuracy: 0.2425
Epoch 4/12
100/100 [==============================] - 114s 1s/step - loss: 0.3823 - accuracy: 0.7419 - val_loss: 4.4443 - val_accuracy: 0.3025
Epoch 5/12
100/100 [==============================] - 113s 1s/step - loss: 0.3786 - accuracy: 0.7453 - val_loss: 3.3136 - val_accuracy: 0.4600
Epoch 6/12
100/100 [==============================] - 115s 1s/step - loss: 0.3697 - accuracy: 0.7519 - val_loss: 1.5114 - val_accuracy: 0.5188
Epoch 7/12
100/100 [==============================] - 114s 1s/step - loss: 0.3736 - accuracy: 0.7478 - val_loss: 0.4601 - val_accuracy: 0.7050
Epoch 8/12
100/100 [==============================] - 116s 1s/step - loss: 0.3747 - accuracy: 0.7550 - val_loss: 0.3814 - val_accuracy: 0.7450
Epoch 9/12
100/100 [==============================] - 114s 1s/step - loss: 0.3764 - accuracy: 0.7491 - val_loss: 0.3649 - val_accuracy: 0.7387
Epoch 10/12
100/100 [==============================] - 114s 1s/step - loss: 0.3744 - accuracy: 0.7497 - val_loss: 0.3541 - val_accuracy: 0.7525
Epoch 11/12
100/100 [==============================] - 115s 1s/step - loss: 0.3696 - accuracy: 0.7553 - val_loss: 0.3515 - val_accuracy: 0.7613
Epoch 12/12
100/100 [==============================] - 115s 1s/step - loss: 0.3690 - accuracy: 0.7450 - val_loss: 0.3766 - val_accuracy: 0.7225
```




```
Log-loss (cost function):
training   (min:    0.367, max:    0.560, cur:    0.369)
validation (min:    0.371, max:    6.650, cur:    0.371)

Accuracy:
training   (min:    0.712, max:    0.762, cur:    0.762)
validation (min:    0.250, max:    0.751, cur:    0.751)
100/100 [==============================] - 169s 2s/step - loss: 0.3685 - acc: 0.7625 - val_loss: 0.3705 - val_acc: 0.7513
```

Let's now evaluate our model
Its accuracy is around 74% which was considered a benchmark back in 2017 when this dataset was launched in a SETI hackathon.
Let's build a confusion matrix to analyse it better:

Task 8: Model Evaluation

In [15]:
```
model.evaluate(x_val, y_val)
```

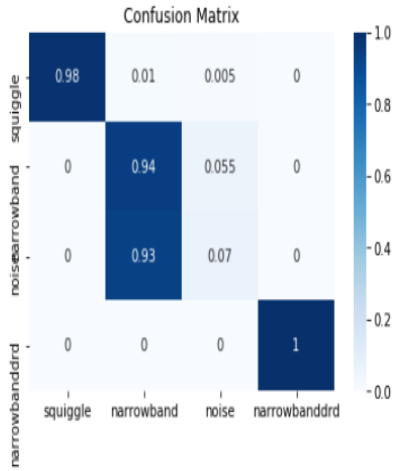
```
800/800 [==============================] - 5s 6ms/sample - loss: 0.3700 - acc: 0.7500
```

Out[15]:
```
[0.3700017468823353, 0.75]
```

In [16]:
```
from sklearn.metrics import confusion_matrix
from sklearn import metrics
import seaborn as sns

y_true = np.argmax(y_val, 1)
y_pred = np.argmax(model.predict(x_val), 1)
print(metrics.classification_report(y_true, y_pred))
print("Classification accuracy: %0.6f" % metrics.accuracy_score(y_true, y_pred))
```

```
              precision    recall  f1-score   support

           0       1.00      0.98      0.99       200
           1       0.50      0.94      0.66       200
           2       0.54      0.07      0.12       200
           3       1.00      1.00      1.00       200

    accuracy                           0.75       800
   macro avg       0.76      0.75      0.69       800
weighted avg       0.76      0.75      0.69       800

Classification accuracy: 0.750000
```

In [20]:
```
labels = ["squiggle", "narrowband", "noise", "narrowbanddrd"]

ax= plt.subplot()
sns.heatmap(metrics.confusion_matrix(y_true, y_pred, normalize='true'), annot=True, ax = ax, cmap=plt.cm.Blues); #annot=True to c

# labels, title and ticks
ax.set_title('Confusion Matrix');
ax.xaxis.set_ticklabels(labels); ax.yaxis.set_ticklabels(labels);
```



In [ ]:

This was a basic model and it achieved an accuracy of around 74% , you can definitely increase its accuracy by using transfer learning and fine tune it on pre-trained CNN networks such as VGG and Image-net.

