# Learning to Deep Learn using Python, Keras, Theano, TensorFlow and a GPU

*Jonathon Hare, 8th March 2017* *(https://github.com/jonhare/os-deep-learning-labs)*

## Acknowledgements

## Introduction

Now we've seen how we can use Keras to work towards the solution of a handwriting recognition problem, we'll turn out focus to OS specific data. The USB key you have been provided with contains the data we'll be using. More specifically, the USB key contains 3-band 25cm resolution orthorectified images covering a sample of SU41, coupled with corresponding 16-bit 1-band (greylevel) images that were created by rasterising the MasterMap vector layer on the basis of the labelled theme attributes of the polygons representing TopographicArea features. The software for generating these rasters is available in the ImageLearn source repository.

We'll use this OS derived data to train some deep models that predict the theme of a pixel, and start to explore commonly used approaches such as transfer learning and fine-tuning of pre-trained networks.

Through this part of the tutorial you'll learn how to:

- How to load patches from the OS data into memory
- How to develop and evaluate a simple CNN for theme classification.
- How to use custom callbacks to monitor training progress.
- How to stitch together classifications for overlapping tiles into a map.
- How to load a pre-trained model and use it to make classifications.
- How to modify and fine-tune a pre-trained model to solve the theme classification problem using OS data.
- How to extract *semantic* features that can be used for transfer learning and finding similar features.

## Prerequisites

As with part 1 of the tutorial, you'll use Python 2 language the keras. We'll also again be using the scikit-learn and numpy packages.

You'll need access to a computer with the following installed:

- Python (> 2.6)
- keras (>= 1.0.0)
- theano (>= 0.8)
- tensorflow (>= 0.11)
- NumPy (>= 1.6.1)
- SciPy (>= 0.9)
- scikit-learn (>= 0.17.0)
- opencv
- pillow (>=4.0.0)

## Getting started

If you haven't already, you need to fork and clone the tutorial code from https://github.com/jonhare/os-deep-learning-labs (https://github.com/jonhare/os-deep-learning-labs) in order to access the provided utility functions that facilitate the use of the OS data. Once you have a local copy of the repository, copy the data directory from the memory stick into the part2 folder of the clone.

We'll start by exploring the data, and look at how we can get that data loaded into memory through python code. If you open the data directory you should see three folders:

```
- The `3band` folder contains the 25cm orthorectified RGB imagery for SU41.
- The `theme` folder contains the rasterised "theme maps" corresponding to each RGB SU41 tile.
- The `key-theme.txt` file contains the mapping from human-readable theme names to the 16-bit greylevel values used in the theme maps to
 represent regions with a particular theme label.
```

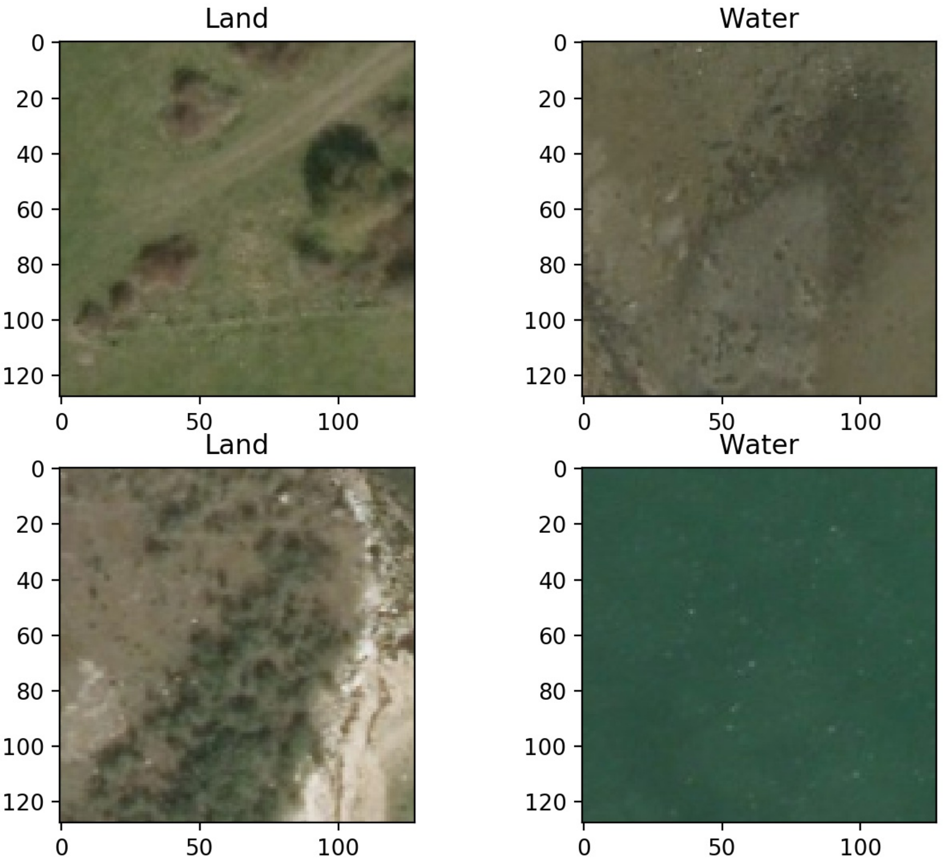
Let's now write a script to use the functions in the provided utils.py file to load some of the data. Unlike the mnist data we explored earlier, we cannot practically load all of the data into memory in one go. We also need to do some processing because, at least initially, we want to extract patches of the larger RGB images together with a label that describes the theme of the central pixel. Three key functions are provided by the utils.py file:

```
- `load_labelled_patches()`: this loads a subset of the data into memory as a tuple containing two numpy arrays; the first contains the patch
es, and the second contains the (one-hot encoded) theme labels.
- `generate_labelled_patches()`: this is a python generator that dynamically returns an a batch of patches and labels. The generator called o
ver and over and will return a new batch of data each time by sampling the underlying 3-band images and theme maps. The generator mak
es it possible to effectively pass over the entire dataset a little bit at a time.
- `load_class_names()`: This loads a list of class names that correspond with the encoded labels produced by the above functions. As the la
bels are one-hot encoded, in order to lookup a class from this list from the one-hot vector, you'll need to use the `argmax()` function on the
 one-hot vector to find the position of the largest value, which maps directly to the list of class names (see below for an example).
```

To demonstrate the load_labelled_patches() function, the following code demonstrates the loading of data by plotting 4 random images and their theme labels:

```
# Plot ad hoc OS data instances
from utils import load_labelled_patches, load_class_names
import matplotlib.pyplot as plt
from keras import backend as K

# load 4 randomly selected 128x128 patches and their labels
(X, y) = load_labelled_patches(["SU4010"], 128, limit=4, shuffle=True)

# load the list of possible labels
clznames = load_class_names()

#if we're using the theano backend, we need to change indexing order for matplotlib to interpret the patches:
if K.image_dim_ordering() == 'th':
  X = X.transpose(0, 3, 1, 2)

# plot 4 images
plt.subplot(221).set_title(clznames[y[0].argmax()])
plt.imshow(X[0])
plt.subplot(222).set_title(clznames[y[1].argmax()])
plt.imshow(X[1])
plt.subplot(223).set_title(clznames[y[2].argmax()])
plt.imshow(X[2])
plt.subplot(224).set_title(clznames[y[3].argmax()])
plt.imshow(X[3])

# show the plot
plt.show()
```

You can see that accessing the dataset is quite easy: the first argument to the load_labelled_patches function specifies a list of tiles to load from; the second specifies how big the patches are, the optional limit argument ensures we only load 4 patches, and the optional shuffle argument tells the function to pick the patches randomly rather than in scan order. Running the above example, you should see something like the image below (obviously you'll get different patches because of the shuffling).



*Exercise: Have a play with the above code and explore the other parameters of the load_labelled_patches function. What happens when you disable shuffle and alter the step size?*

## A simple CNN for theme classification

Now let's try something a little more challenging and take our *larger* convolutional network from the experiments with mnist and apply it to the problem of theme classification. Firstly we need to setup the data for training (this time using the generator so we don't have to worry about memory usage), and it would also be sensible to load a smaller amount of data into memory for monitoring validation performance during training:

```
import keras
import matplotlib.pylab as plt
from utils import generate_labelled_patches, load_labelled_patches, load_class_names

# define the patch size as a variable so its easier to change later. For now,
# we'll set it to 28, just like the mnist images
patch_size = 28

# load data
train_data = generate_labelled_patches(["SU4010"], patch_size, shuffle=True)
valid_data = load_labelled_patches(["SU4011"], patch_size, limit=1000, shuffle=True)

# load the class names
clznames = load_class_names()
num_classes = len(clznames)
```

Note that we've loaded the training and validation data from different tiles to keep things fair. Now we can add the network definition from part 1. We'll make a slight change to the previous `larger_model()` function so that it allows us to specify the input and output sizes, and we'll also pull out the compile statement as it would be the same for many model architectures:

```
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from keras.layers import Flatten
from keras.layers.convolutional import Convolution2D
from keras.layers.convolutional import MaxPooling2D

def larger_model(input_shape, num_classes):
    # create model
    model = Sequential()
    model.add(Convolution2D(30, 5, 5, border_mode='valid', input_shape=input_shape, activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Convolution2D(15, 3, 3, activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout(0.2))
    model.add(Flatten())
    model.add(Dense(128, activation='relu'))
    model.add(Dense(50, activation='relu'))
    model.add(Dense(num_classes, activation='softmax'))

    return model

# build the model
model = larger_model(valid_data[0][0].shape, num_classes)
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

Specifying the input shape using the shape of the first validation instance allows us to avoid having to worry about how the backend is storing the patches (if we were using theano conventions it would be `(3,patch_size,patch_size)`), whereas with tensorflow it would be `(patch_size,patch_size,3)`. We're now in a position to add the code to fit the model. Because this time we're loading the data using a generator rather than statically we use the `fit_generator()` method instead of `fit`:

```
# Fit the model
model.fit_generator(train_data, samples_per_epoch=10016, nb_epoch=10, validation_data=valid_data, verbose=1)
```

We've specified 10016 samples_per_epoch to keep computation time down; this number is exactly divisable by the default batch size of 32. In actuallity if we want to sample all patches from a single tile in an epoch (assuming a 1px step), there are (4000-patch_size)**2 samples available to us. This is a rather large number and will take a significant amount of time - in fact for reasonable patch sizes this is approaching the total size of the ImageNet data from just one tile! Note that by having a smaller number of samples per epoch than in the actual data that each epoch will end up having a different sample of training data to work with.

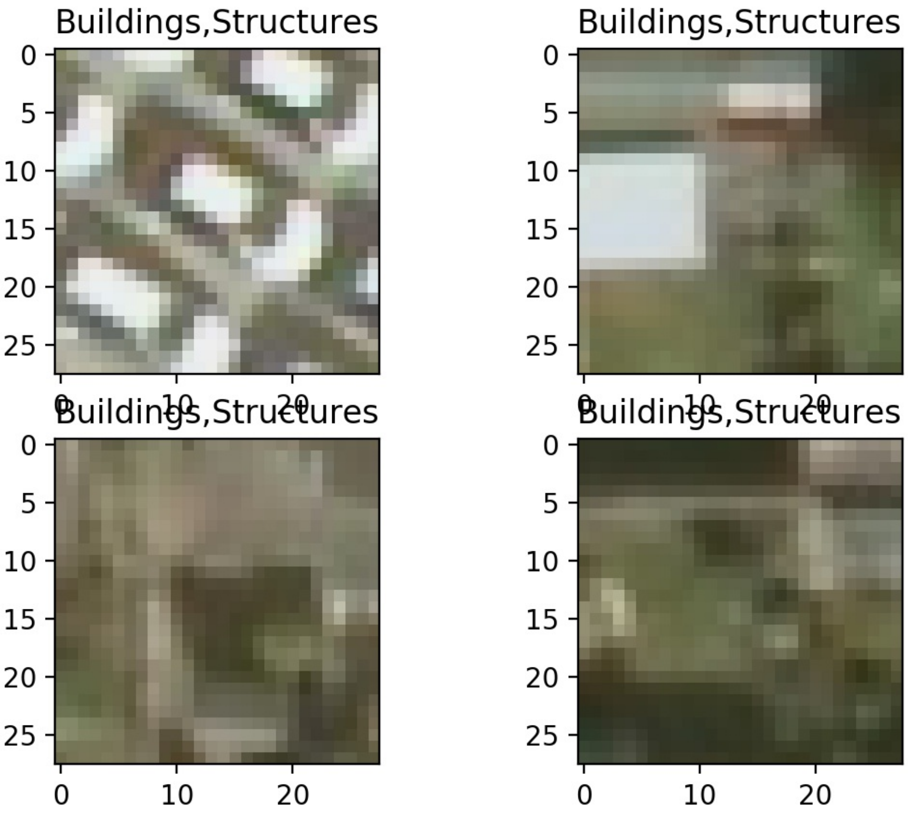
Finally, before we try running this model, lets add the code to load samples from another tile and make and display some classifications:

```
from keras import backend as K

# load 4 randomly selected patches and their labels
(X_test, y_test_true) = load_labelled_patches(["SU4012"], patch_size, limit=4, shuffle=True)
y_test = model.predict_classes(X_test)

#if we're using the theano backend, we need to change indexing order for matplotlib to interpret the patches:
if K.image_dim_ordering() == 'th':
    X_test = X_test.transpose(0, 3, 1, 2)

# plot 4 images
plt.subplot(221).set_title(clznames[y_test[0].argmax()])
plt.imshow(X_test[0])
plt.subplot(222).set_title(clznames[y_test[1].argmax()])
plt.imshow(X_test[1])
plt.subplot(223).set_title(clznames[y_test[2].argmax()])
plt.imshow(X_test[2])
plt.subplot(224).set_title(clznames[y_test[3].argmax()])
plt.imshow(X_test[3])

# show the plot
plt.show()
```

Running this should result in the following:

```
Using TensorFlow backend.
Epoch 1/10
10016/10016 [==============================] - 25s - loss: 0.9179 - acc: 0.7318 - val_loss: 0.6572 - val_acc: 0.8310
Epoch 2/10
10016/10016 [==============================] - 7s - loss: 0.6426 - acc: 0.8190 - val_loss: 0.6077 - val_acc: 0.8490
Epoch 3/10
10016/10016 [==============================] - 7s - loss: 0.6037 - acc: 0.8259 - val_loss: 0.5387 - val_acc: 0.8500
Epoch 4/10
10016/10016 [==============================] - 7s - loss: 0.5786 - acc: 0.8242 - val_loss: 0.5549 - val_acc: 0.8550
Epoch 5/10
10016/10016 [==============================] - 7s - loss: 0.5646 - acc: 0.8359 - val_loss: 0.5586 - val_acc: 0.8520
Epoch 6/10
10016/10016 [==============================] - 7s - loss: 0.5355 - acc: 0.8404 - val_loss: 0.5150 - val_acc: 0.8560
Epoch 7/10
10016/10016 [==============================] - 7s - loss: 0.5466 - acc: 0.8334 - val_loss: 0.4932 - val_acc: 0.8550
Epoch 8/10
10016/10016 [==============================] - 7s - loss: 0.5469 - acc: 0.8352 - val_loss: 0.5890 - val_acc: 0.8580
Epoch 9/10
10016/10016 [==============================] - 7s - loss: 0.5180 - acc: 0.8427 - val_loss: 0.5248 - val_acc: 0.8610
Epoch 10/10
10016/10016 [==============================] - 7s - loss: 0.5025 - acc: 0.8481 - val_loss: 0.5215 - val_acc: 0.8550
```

with a set of sample predictions that looks something like this:



In this particular case the overall accuracies are all quite high (in terms of both training and validation), which is pleasing. Be aware though that we're using a relatively small set of both training and validation data, and that there is a very high bias in the class distribution which inevitably could lead to higher accuracies because of common classes.

> *Exercise: Have a play with the above code and explore the effect of patch size and the amount of training and validation data.*

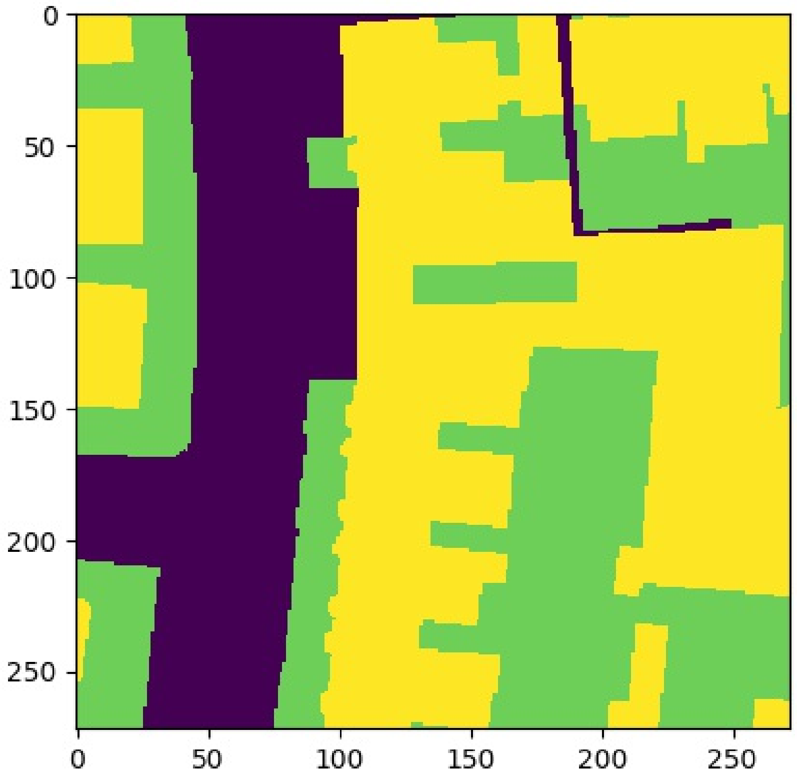
## Mapping the classifications

Now we can make predictions for a tile it would be quite nice to make a modification so that we can try and reconstruct theme maps directly from the 3-band images. This is often called semantic segmentation. An easy approach that we can take is to take spatiatlly sequential patches, and reconstruct an "image" based on the class assignments. This is effective, but highly computationally inefficient - better approaches are to convert the network to be fully convolutional (see e.g. https://devblogs.nvidia.com/parallelforall/image-segmentation-using-digits-5/#comment-1891 (https://devblogs.nvidia.com/parallelforall/image-segmentation-using-digits-5/#comment-1891) ), or to use a specialist type of network that actually outputs segmentation maps directly (usually these networks are based on layers of downsampling convolutions like we're using in our network, followed by layers of upsampling or deconvolution which aim to increase the spatial resolution back up to the size of the input).

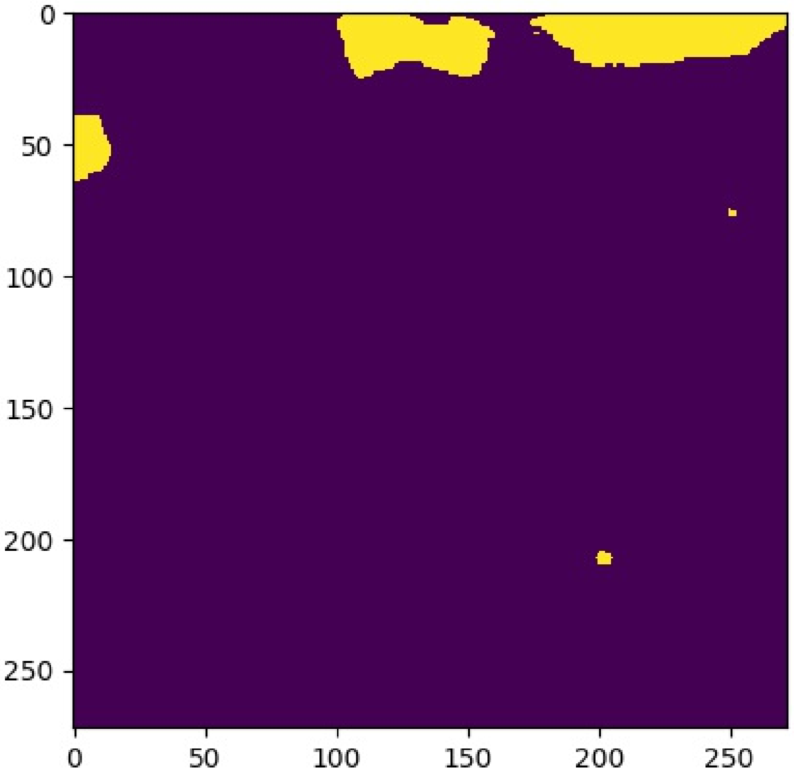
For now, let's implement the naive approach by building a theme map for some test data. We'll keep the same code as above, but modify the parts involving the test data as follows:

```
import numpy as np

# load some test data; this time we specifically load patches from a 300x300 square of a tile in scan-order
test_data = load_labelled_patches(["SU4012"], patch_size, subcoords=((0,0), (300,300)))

# we can reshape the test data labels back into an image and save it
tmp = np.zeros(test_data[1].shape[0])
for x in xrange(0, test_data[1].shape[0]):
    tmp[x] = test_data[1][x].argmax()
tmp = tmp.reshape((300-patch_size, 300-patch_size))
plt.figure()
plt.imshow(tmp)
plt.savefig("test_gt.png")

# and we can do the same for the predictions
clzs = model.predict_classes(test_data[0])
clzs = clzs.reshape((300-patch_size, 300-patch_size))
plt.figure()
plt.imshow(clzs)
plt.savefig("test_pred.png")
```

If we now run this, we'll get a "ground-truth" image that looks like this:



and a "predictions" image that looks something like this (results will vary depending on the sample of data used for training):



In this case we can see that the result is not particularly good, although undoubtly this is due to the tiny amount of training our network has had as well as the fact that our predictions are entirely based on looking at a 28x28 pixel window of the image (bearing in mind that this is only 7m x 7m on the ground, which is pretty tiny and obviously fails to capture any context from the surroundings).

One additional change we might like to make to our code is to add a “callback” that produces and saves a theme-map for the validation data after each epoch; this will allow us to visually monitor how the network is learning. Firstly we need to modify the validation data to be loaded from a region of a tile in scan order rather than from random sampling:

```
valid_data = load_labelled_patches(["SU4011"], patch_size, subcoords=((0,0), (300,300)))
```

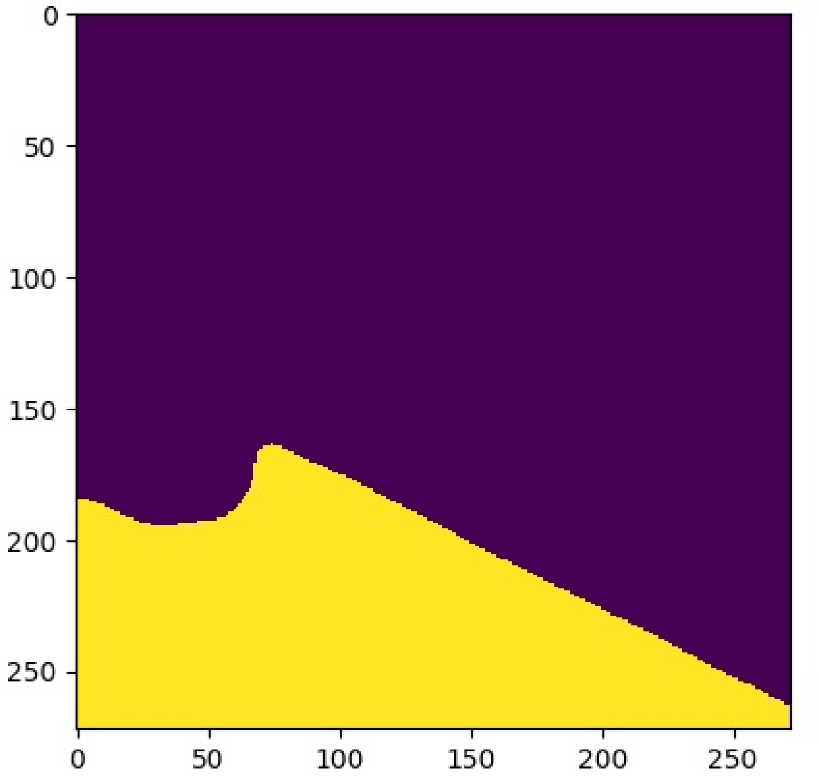
Next we need to define the callback by extending the `keras.callbacks.Callback` class, and implement the `on_epoch_end` method to make predictions on the validation data, reformat them into an image and save the result:

```
class DisplayMap(keras.callbacks.Callback):
    def on_epoch_end(self, epoch, logs={}):
        clzs = model.predict_classes(valid_data[0])
        clzs = clzs.reshape((300-patch_size, 300-patch_size))
        plt.figure()
        plt.imshow(clzs)
        plt.savefig("map_epoch%s.png" % epoch)
```

Finally, we need to modify the call to `fit_generator` to include the callback:

```
model.fit_generator(train_data, samples_per_epoch=10016, nb_epoch=10, validation_data=valid_data, verbose=1, callbacks=[DisplayMap()])
```

If we run the code now after each epoch has passed an image will be saved. Here's the image from the first epoch:



*Exercise: It's a little difficult to interprete whether the above validation theme map is actually any good because we don't exactly know what it should look like (we can compare against the ground-truth in the data directory, but this is a different size). Add some additional code to save the validation data ground-truth theme map before training starts so we have something to compare against.*

## Using a better network model - transferring and finetuning a pretrained ResNet

Training a network from scratch can be a lot of work. Is there some way we could take an existing network trained on some data with one set of labels, and adapt it to work on a different data set with different labels? Assuming that the inputs of the network are equivalent (for example, image with the same number of bands and size), then the answer is an emphatic yes! This process of "finetuning" a pre-trained network has become common-place as its much faster an easier than starting from scratch.

Let's try this in practice - we'll start by loading a pre-trained network architecture called a Deep Residual Network (or ResNet for short) that has been trained on the 1000-class ImageNet dataset. The ResNet architecture is very deep - it has many (in our case 50) convolutional layers and is currently one of the best performing architectures on the ImageNet challenge. The tutorial git repo contains code that implements the resnet50 architecture, and automatically downloads the pre-trained model weights. We'll start by using this to load the model and test it by classifying an image:

```
from resnet50 import ResNet50
from keras.preprocessing import image
from imagenet_utils import decode_predictions, preprocess_input
import numpy as np

model = ResNet50(include_top=True, weights='imagenet')

img_path = 'images/mf.jpg'
img = image.load_img(img_path, target_size=(224, 224))
x = image.img_to_array(img)
x = np.expand_dims(x, axis=0)
x = preprocess_input(x)

preds = model.predict(x)
print('Predicted:', decode_predictions(preds))
```

If we run this (it will take a little longer the first time as the model is downloaded), it should print the following:

```
Using TensorFlow backend.
K.image_dim_ordering: tf
('Predicted:', [[(u'n02640242', u'sturgeon', 0.35990164), (u'n02641379', u'gar', 0.3399567), (u'n02514041', u'barracouta', 0.26639012), (u'n0
2536864', u'coho', 0.028537149), (u'n01484850', u'great_white_shark', 0.0025088955)]])
```

Indicating that our input image was likely to contain a fish!

*Exercise: try the model with some of your own images*

We're now in a position to start to hack the model structure. Fundamentally we need to first remove the classification layer at the end of the model and replace it with a new one (with a different number of classes):

```
def hack_resnet(num_classes):
    model = ResNet50(include_top=True, weights='imagenet')

    # Get input
    new_input = model.input
    # Find the layer to connect
    hidden_layer = model.layers[-2].output
    # Connect a new layer on it
    new_output = Dense(num_classes) (hidden_layer)
    # Build a new model
    newmodel = Model(new_input, new_output)

    return newmodel
```

The actual process of finetuning involves us now training the model with our own data. This is as simple as compiling the model and running the fit or fit_generator methods as before. As the network is already largely trained, we'll likely want to use a small learning rate so not to make big changes in weights:

```
model.compile(loss='binary_crossentropy',
              optimizer=optimizers.SGD(lr=1e-4, momentum=0.9),
              metrics=['accuracy'])
```

Often we'll first "freeze" the weights of the already trained layers whilst we learn initial weights for our new layer to avoid overfitting before training:

```
# set weights in all but last layer
# to non-trainable (weights will not be updated)
for layer in model.layers[:len(model.layers)-2]:
    layer.trainable = False
```

If we have lots of training data we could then unlock these layers and perform end-to-end finetuning afterwards. The Standford CS231n course pages have lots of useful hints on fine-tuning: http://cs231n.github.io/transfer-learning/ (http://cs231n.github.io/transfer-learning/)

> *Exercise: try finetuning the resnet50 with the theme data. You'll need a GPU to do this effectively as it's very slow!*

## Extracting features from a model

Sometimes you want to do things that are not so easily accomplished with a deep network. You might want to build classifiers using very small amounts of data, or you might want a way of finding things in photographs that are in some way semantically similar, but don't have exactly the same classes. CNNs can actually help here using a technique known often called transfer learning (and related to the fine tuning that we just looked at). If we assume we have a trained network, then by extracting vectors from the layers before the final classifier we should have a means of achieving these tasks, as the vectors are likely to strongly encode semantic information about the content of the input image. If we wanted to quickly train new classifiers for new classes, we could for instance just use a relatively simple linear classifier trained on these vectors. If we wanted to find semantically similar images, we could just compare the Euclidean distance of these vectors.

Keras makes it pretty easy to get these vector representations. First we have to remove the end of out network to the point we with to extract the features - for example in the resnet we might want features from the final flatten layer before the final dense connections (this gives us a 2048 dimensional vector from every 224x224 dimensional inout image):

```
model = ResNet50(include_top=True, weights='imagenet')

# Get input
new_input = model.input

# Find the layer to end on
new_output = model.layers[-2].output

# Build a new model
newmodel = Model(new_input, new_output)
```

With this model, we can use the `predict()` method to extract the features for some inputs. To demonstrate, we can put the whole thing together and generate some vectors from some samples of our 3-band images

```
from resnet50 import ResNet50
from imagenet_utils import preprocess_input
from keras.models import Model
from utils import load_labelled_patches


model = ResNet50(include_top=True, weights='imagenet')

# Get input
new_input = model.input

# Find the layer to end on
new_output = model.layers[-2].output

# Build a new model
newmodel = Model(new_input, new_output)

(X, y_test_true) = load_labelled_patches(["SU4012"], 224, limit=4, shuffle=True)
X = preprocess_input(X)

features = newmodel.predict(X)

print features.shape
print features
```

(Obviously this will be more effective if the network has been trained or fine-tuned on the same kind of data that we're extracting features from.)

> *Exercise: try generating some features for different patches and calculating the Euclidean distances between these features. How do the Euclidean distances compare to your perception of similarity between the patches?*