

# Machine Learning Engineer Nanodegree

## Capstone Project

David Howard

10/23/2017

## I. Definition

### Project Overview

SLAM (simultaneous location and mapping) is a well-known technique for using sensor data to create a map of a geographic region and also keep track of the sensors location within the constructed map. Typically this is done with either a particle filter or a Kalman filter. One application of SLAM operates on an unknown region and builds up knowledge about the region via sensor measurements. Another application uses existing knowledge, such as maps, along with sensors, and is aimed at determining the sensor's location within the known region. [1]

[https://en.wikipedia.org/wiki/Simultaneous\\_localization\\_and\\_mapping](https://en.wikipedia.org/wiki/Simultaneous_localization_and_mapping))

In reading about SLAM, I found it interesting that I had previously been trained to be a human SLAM algorithm when I was a helicopter pilot in the U.S. Army. We trained to navigate high speed low level flight by using only a topographic map and our eyes observing the terrain elevation features (no GPS allowed). There is a name for this technique, Nap-Of-the-Earth flight. [2]

[https://en.wikipedia.org/wiki/Nap-of-the-earth#Helicopter\\_NOE\\_flying](https://en.wikipedia.org/wiki/Nap-of-the-earth#Helicopter_NOE_flying))

One source of potential data for use in a SLAM system is a digital elevation model, generically referred to as DEM. A great source of such data was created by the NASA Shuttle Radar Topography Mission (SRTM). In 2000, the Space Shuttle was used to create a digital elevation map of the entire world (minus some polar regions). The NASA SRTM data is publicly available for download, in various height resolutions. [3] (<https://www2.jpl.nasa.gov/srtm/mission.htm>)

### Problem Statement

This project will attempt to create a simple SLAM type implementation using a convolutional neural network to determine position within a known region. The train/test data used for learning will be SRTM elevation data treated as an image. Input for evaluation will be images composed of pieces of the same data with possible modifications such as distortion, alignment and reduced resolution. Output will be a predicted position of the input images.

### Metrics

The outputs of the model will be predictions of the probabilities of a test image location matching one of the actual locations. The outputs will be a set of probabilities for each test image.

The models are evaluated by comparison of the categorical\_crossentropy loss of the number of epochs run. Tensorboard graphs will be used to visualize and compare the loss over various iterations of the models.

The 'absolute' accuracy will be the average of the highest predicted probability being correct over the set of test data as was computed in the dog project.

$\text{test\_accuracy} = 100 * \text{Sum}(\text{test predictions} / \text{test labels})$

## II. Analysis

### Data Exploration

SRTM data is available online at [4] (<https://dds.cr.usgs.gov/srtm>).

SRTM data is available in multiple resolutions. The data is segmented into 1 latitude/longitude degree squares. The highest publicly available resolution is 1 arc-second per elevation posting (Level 2, ~30 meters at the equator), which results in a 3600x3600 matrix of elevations. The data files are actually 3601x3601 in order to fill overlaps if multiple cells are composed together. Other resolutions are 3 arc seconds (Level 1, ~90 meters) and (Level 0, 30 arc seconds). A detailed description of the data is available at [5] ([https://dds.cr.usgs.gov/srtm/version2\\\_\\_1/Documentation/SRTM\\\_\\_Topo.pdf](https://dds.cr.usgs.gov/srtm/version2\__1/Documentation/SRTM\__Topo.pdf))

Organization of data

- Level 1
  - Binary file labeled type '.hgt'
  - 1201x1201 grid of height postings
  - Covers 1-degree latitude/longitude
  - Post spacing is 3 arc-seconds
  - Data type is unsigned 16 bit, big-endian
  - Rows are lower to higher
  - Columns are left to right
  - Naming convention specifies location e.g. N39W120.hgt
- Level 2
  - Binary file labeled type '.hgt'
  - 3601x3601 grid of height postings
  - Covers 1-degree latitude/longitude
  - Post Spacing is 1 arc-seconds
  - Data type is unsigned 16 bit, big-endian
  - Rows are lower to higher
  - Columns are left to right
  - Naming convention specifies location e.g. N39W120.hgt

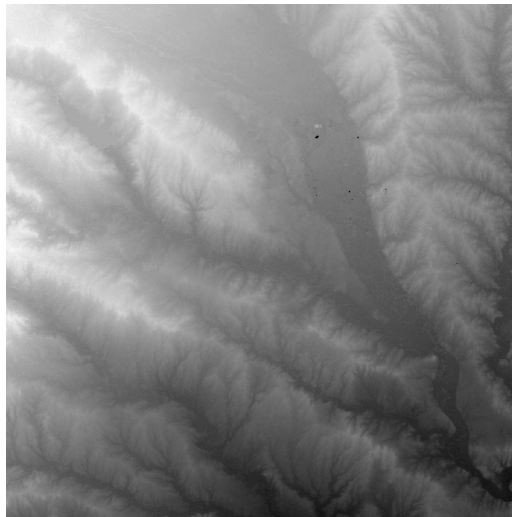
Note: I found that during testing that the level 2 data used too much GPU memory on my local machine and that training times were too long (\$\$) on a Google Compute Engine with K80 GPU, even on small models. As a result only the level 1 data is used in testing. The images covered are the same, just at a lower resolution.

## Exploratory Visualization

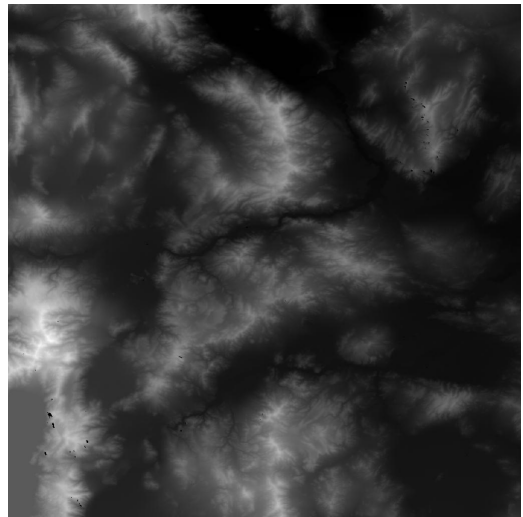
The data is in a non-image binary format but can be converted to displayable images for visualization by functions in the included 'srtm' module. In the following discussions the term 'image' is used interchangeably to refer to either the raw .hgt data or a resulting displayable image, since these two formats can be converted from one to the other. Because the data is single values per elevation posting, the resulting images can all be treated as monochrome.

### Baseline Data as Images

When converting the .hgt file to an image, the data is normalized to the range [0..255] to increase the contrast between points of lower and higher elevation.



N37W098 (Wichita KS area, flat)



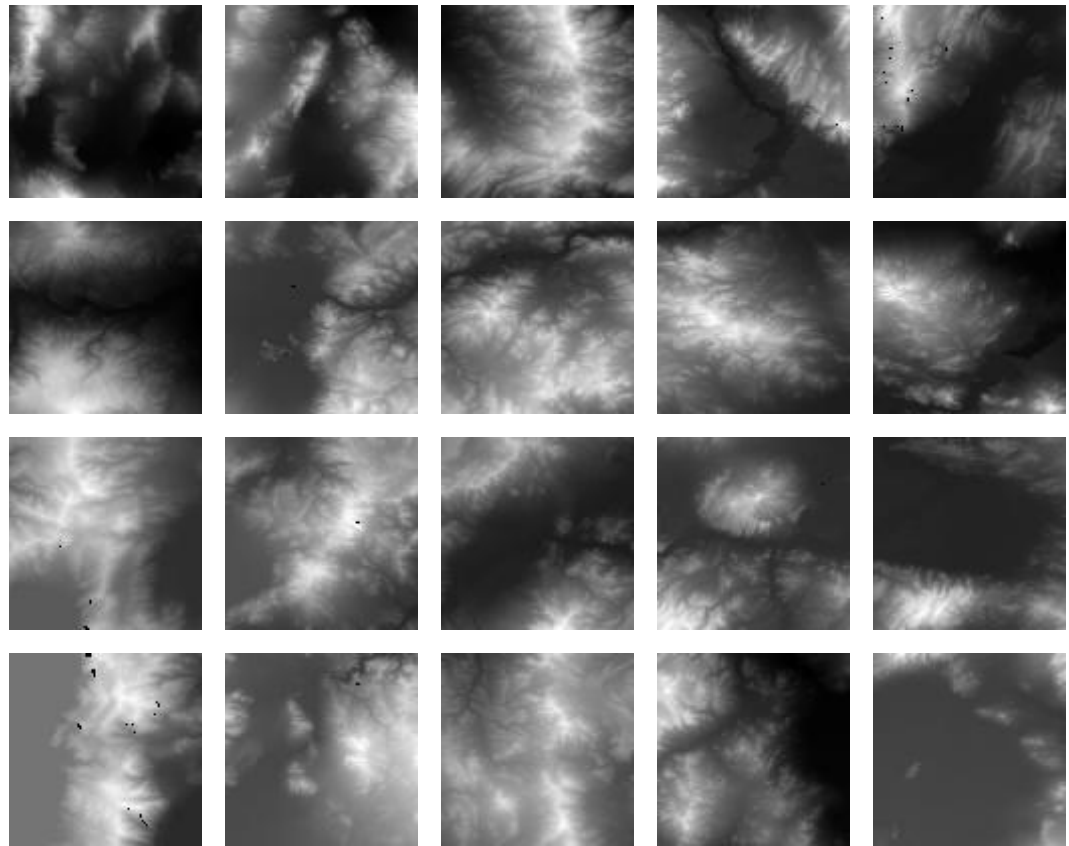
N39W120 (Reno NV area, mountainous)

### Subdivision of input data

The full images will be subdivided into an  $N \times N$  set of squares that will be the input to the model for training and test. The number of subdivisions will be determined by experimentation, based on training time and results. The smaller the subdivided images, the greater the resolution of predicted location. However, if the subdivided images are too small, it may result in less accurate results because of fewer distinguishing features between similar images. The labels will be integer numbers from 0.. $N \times N$  starting at the upper left subdivision, row major order. These actual latitude/longitude location can be determined from the label and subdivision level.

Example of input data subdivision N39W120, 5x5 array.



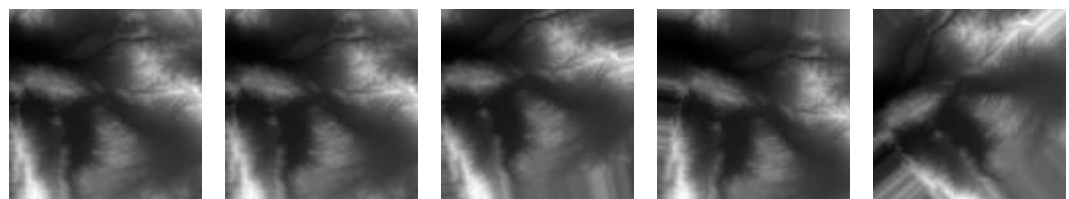


### Augmentation of input data

The subdividing process results in only one image and label for each data point. After subdivision, each resulting Training with just one image per label is insufficient to achieve good learning results. It can result in an overfit model that could only identify a location if the image is nearly an exact duplicate and orientation. To overcome this, for each subdivided object will be multiplied with distortion using Keras' ImageDataGenerator function. This function creates multiple instances of each subdivided object with various rotations and offsets.

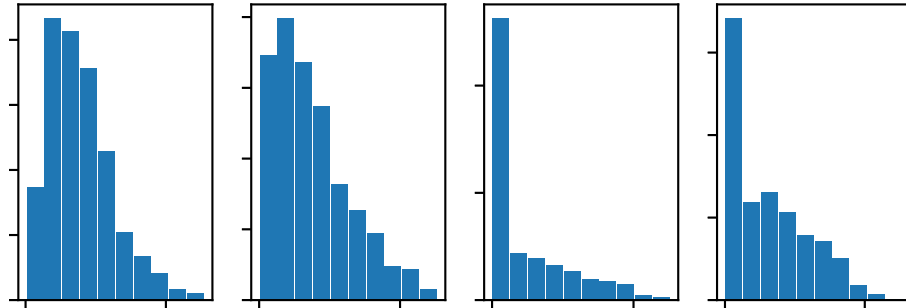
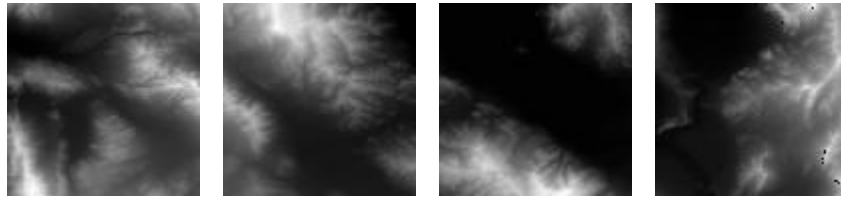
The extent of image distortion used is subject to experimentation. One metric in the report will be distortion parameters vs probability of successful detection.

Example of 4 modifications of one subdivision, upper left corner of N39W120, renormalized and distorted. Image on left is original. ImageDataGen parameters used were rotation=45,height\_shift=0.1,width\_shift=0.1.



### Histograms

Histograms of 4 sample images. X axis is pixel/height to visualize differences in height distribution. Although not directly used, it was helpful to visualize the magnitude of differences between the subdivided images.



## Algorithms and Techniques

The processing algorithms will leverage the techniques and general approach from the previous 'dog project'. The main difference is that the input data files have only one image for each class, rather than many images as were provided in the dog project. This limitation will be mitigated by the use of Keras' feature augmentation to take the unique individual images and vary them to provide sufficient training and test data to train the model. Initially, the feature augmentation hyperparameters will be limited in order to simplify building the model. Once a model is working then the hyperparameters will be tuned to give more variation in the input data.

The primary algorithm used will be a convolutional neural network, implemented using python and Keras using the tensorflow-gpu backend. The model will be iterated until results no further improvement is gained with an acceptable training time.

The model will use some number of Convolutional layers with Dropout and MaxPooling. Variations of kernel size, padding and activation functions will be tried for these layers. The final layer be a Dense layer with GlobalAveragePooling and softmax activation. The model will be compiled with the categorical crossentropy loss function, and with variations of optimizer function evaluated.

The variations of hyperparameters and training time are logged with the resulting evaluation metrics and included in the conclusions of this report.

## Benchmark

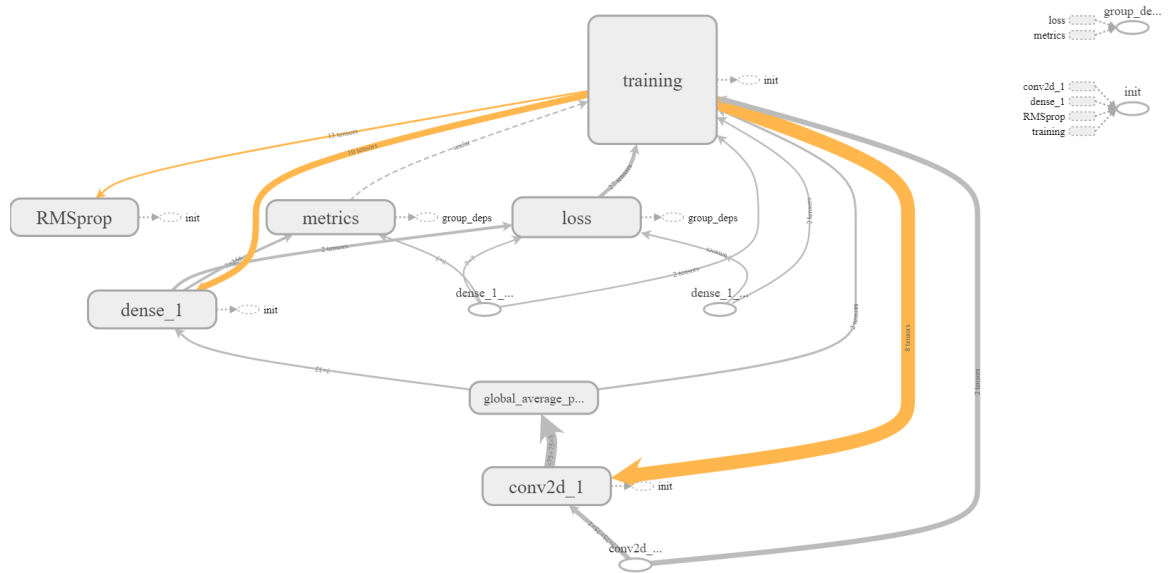
There is no established benchmark for this project. A benchmark is provided by using the same input to a naive 2 level feed-forward neural network. The results of this benchmark will be compared to the results from the final model.

**code**

```
# create model
model = Sequential()
model.add(Conv2D(filters=32, kernel_size=3, padding='same', activation='relu', input_shape=train_X.shape[1:]))
model.add(GlobalAveragePooling2D())
model.add(Dense(labels, activation='softmax'))

# compile the model
model.compile(optimizer="rmsprop",
              loss="categorical_crossentropy",
              metrics=["accuracy"])
```

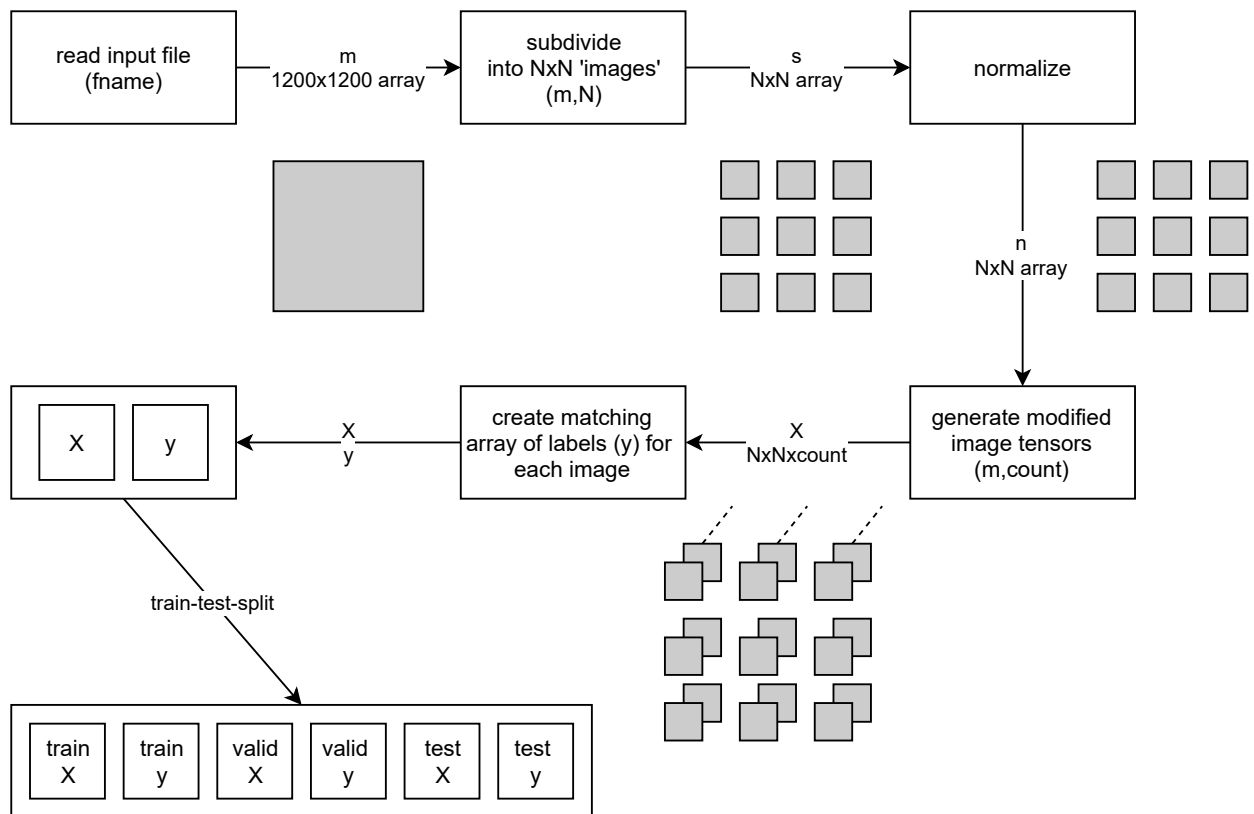
## Model Graph



### III. Methodology

## Data Preprocessing

Processing requires reading an input of 1200x1200 (level 1) or 3600x3600 (level2) input dataset, and producing as output the X and Y



## Implementation

### Initial Setup

The basic structure of the model is based on the structure I used in "Step 3: Create a CNN to Classify Dog Breeds (from Scratch)" with enhancements. Using the simple Benchmark model (which had short training time), I added Tensorboard, History and a custom progress callbacks. I also added printout of additional metrics from the History object, including the final and best values of the loss, accuracy, validation loss and validation accuracy. Graphics of the loss and accuracy were captured from TensorBoard. These metric functions were added to the solution code also.

Note on metrics printout: If final accuracy or final loss is worse than the corresponding best value, then the model results deteriorated in one of more of the final epochs. That is visible in the loss/accuracy graphs also.

### Platform

Test were run on two platforms, depending on memory limitations:

- Intel Core i5, 16GB RAM, Nvidia GTX 1050 GPU with 2GB VRAM
  - 2017-10-21 14:17:29.125682: I tensorflow/core/common\_runtime/gpu/gpu\_device.cc:955] Found device 0 with properties:
    - name: GeForce GTX 1050
    - major: 6 minor: 1 memoryClockRate (GHz) 1.455
    - pciBusID 0000:01:00.0
    - Total memory: 1.95GiB
    - Free memory: 1.18GiB

- 2017-10-21 14:17:29.125711: I tensorflow/core/common\_runtime/gpu/gpu\_device.cc:976] DMA: 0
- 2017-10-21 14:17:29.125718: I tensorflow/core/common\_runtime/gpu/gpu\_device.cc:986] 0: Y
- 2017-10-21 14:17:29.125746: I tensorflow/core/common\_runtime/gpu/gpu\_device.cc:1045] Creating TensorFlow device (/gpu:0)
- (device: 0, name: GeForce GTX 1050, pci bus id: 0000:01:00.0)
- Google Compute Instance with Nvidia K80 GPU
  - 2017-10-22 15:30:11.587454: I tensorflow/core/common\_runtime/gpu/gpu\_device.cc:955] Found device 0 with properties:
  - name: Tesla K80
  - major: 3 minor: 7 memoryClockRate (GHz) 0.8235
  - pciBusID 0000:00:04.0
  - Total memory: 11.17GiB
  - Free memory: 11.09GiB
  - 2017-10-22 15:30:11.587708: I tensorflow/core/common\_runtime/gpu/gpu\_device.cc:976] DMA: 0
  - 2017-10-22 15:30:11.587776: I tensorflow/core/common\_runtime/gpu/gpu\_device.cc:986] 0: Y
  - 2017-10-22 15:30:11.587850: I tensorflow/core/common\_runtime/gpu/gpu\_device.cc:1045] Creating TensorFlow device (/gpu:0)
  - (device: 0, name: Tesla K80, pci bus id: 0000:00:04.0)

## Basic Approach

The initial approach is to start with the Benchmark and add Layers until the results stopped improving or the training time became too long. The procedure was:

- LOOP
  - add a Conv2D layer
  - add Dropout
  - add MaxPooling
  - all Conv2D layers use
    - 'relu' activation function
    - 'rmsprop' optimizer
    - 'categorical\_crossentropy' loss function
    - 'same' padding
  - use 300 epochs (this value seemed to allow all the models to converge)
  - record training time, loss/acc from model.evaluate, and final loss and % accuracy as computed in the dog project

## Refinement

- These refinements were tried on each model (with a given set of layers)
  - testing each iteration of the model with each of 16 and 32 filters
  - testing each iteration of the model with kernel size of 3 and 5



The metrics were evaluated at each iteration of the model and the best combination of filters and kernel size were retained. Then the next layer was added and the refinement repeated on that layer. It is possible that when additional layers are added, that retrying combinations of hyperparameters on the preceding layers could get improvement. However this led to a combinatorial explosion of trial runs so at each addition of a layer, only the last layer was tested with varied hyperparameters.

## Problems

- data set size
  - The difference in number of points for the two different data sets is large
  - level 1 files have 1,440,000 points
  - level 2 files have 12,960,000 points
  - as expected training times on the level 2 data files was substantially longer than on the level 1 files
- GPU memory limitation
  - the local machine used has a 2GB Nvidia GTX 1050
  - failures due to memory exhaustion occurred with the level 2 files and large numbers of layers and filters
  - although the Google Compute Engine K80 GPU has substantial memory, its training times were very long even on small models

## Solutions

- memory limitations
  - the data preprocessing code was cleaned up to free unused memory before running the model. this was effective in allowing more complex models and the level 2 data sets to run successfully
  - runs that experienced failures on the local machine were run on a Google Compute instance with an Nvidia K80 GPU
  - only the 1200x1200 data sets were used due to training times and memory requirements
- initial refinement runs
  - the initial refinement runs were performed locally on the level1 files to reduce training time and avoid memory exhaustion
  - since the level1 and level2 files cover the same geographic area, they will have similar feature sets, with level1 having less resolution.
  - the hypothesis going in is that using level 1 files to refine the model would be sufficient and that final runs could be performed on the level 2 files. The results were compared to confirm if the that hypothesis is correct.
  - when necessary, the level 2 runs were performed on the Google Compute instance instead of locally

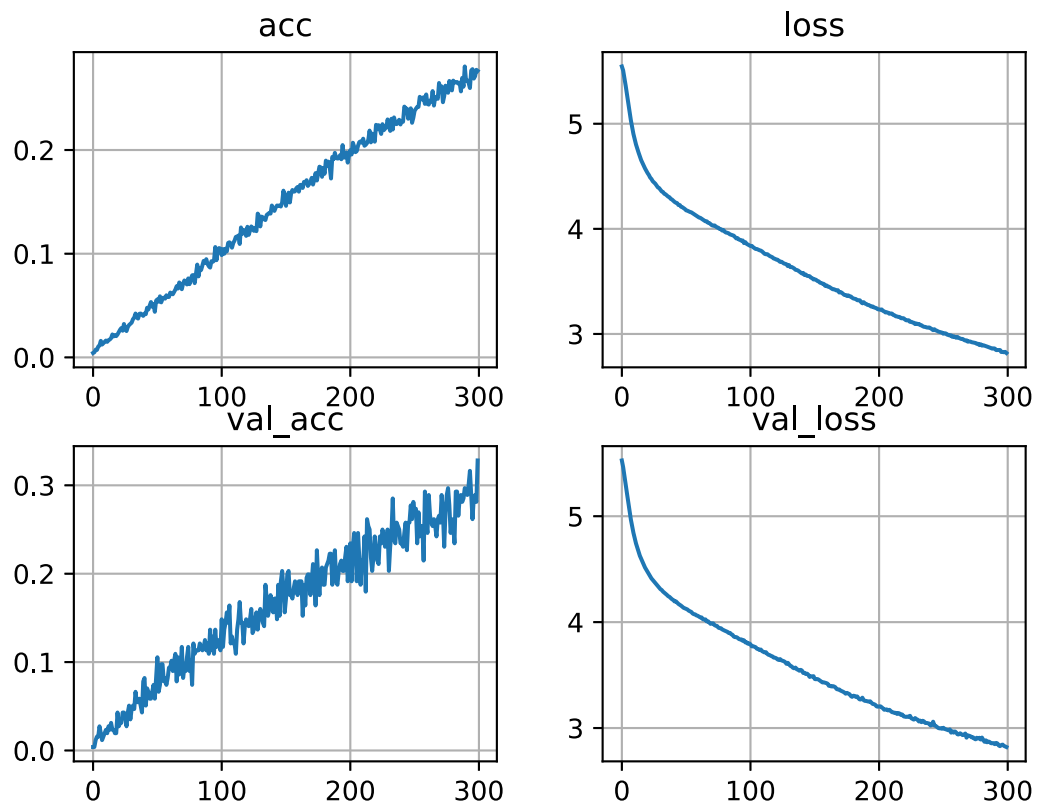
## Level 2 Results For Model 1 on Google Compute with K80 GPU

This result is from level2 3600x3600 data and had a very long training time. This is why the remaining tests were performed with the 1200x1200 data. The level2 data was too large to run on my local machine 1050 GPU.

timestamp : 20171022\_152912  
config : config/model1/level2-N39W120.json  
datafile : data/level2/N37W098.hgt  
divisor : 16  
augments : 15  
epochs : 300  
model : model1  
input shape : (3600, 3600)  
subdivided shape : (256, 225, 225)  
normalized shape : (256, 225, 225)  
X shape : (4096, 225, 225, 1)  
y shape : (4096,)  
...  
epochs : 300  
training time : 76:07

## Benchmark Results

benchmark 20171021\_141717



```

timestamp : 20171023_074451
config    : config/benchmark.json
datafile  : data/level1/N39W120.hgt
divisor   : 16
augments  : 15
epochs    : 300
model     : benchmark
input shape      : (1200, 1200)
subdivided shape : (256, 75, 75)
normalized shape : (256, 75, 75)
X shape         : (4096, 75, 75, 1)
y shape         : (4096,)
model init
train data      X: (3072, 75, 75, 1) y: (3072, 256)
validation data X: (256, 75, 75, 1) y: (256, 256)
test data       X: (256, 75, 75, 1) y: (256, 256)

```

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 75, 75, 32)	320
global_average_pooling2d_1 (	(None, 32)	0
dense_1 (Dense)	(None, 256)	8448
Total params: 8,768		
Trainable params: 8,768		
Non-trainable params: 0		

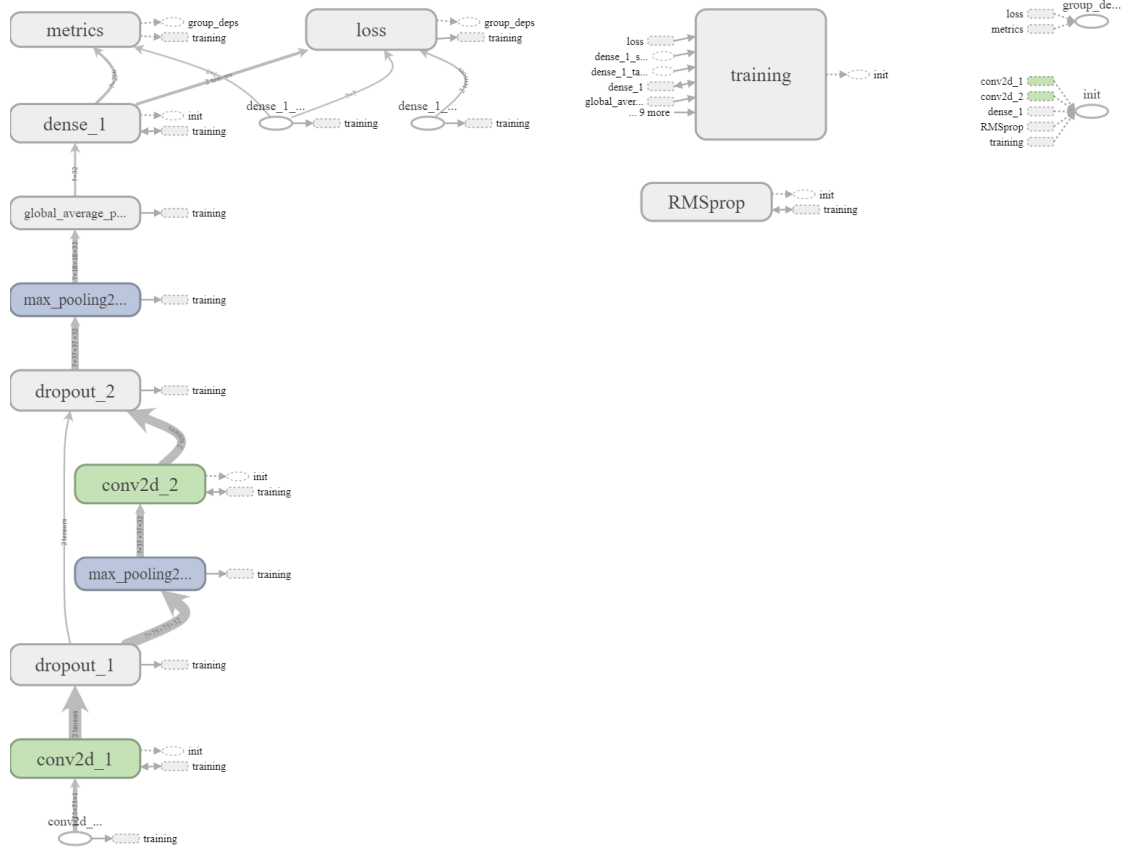
```

loss : 3.4349829852581024
acc  : 0.12890625
best accuracy : 0.1335
final loss    : 3.4889
best loss     : 3.4889
final val acc : 0.1563
best val acc  : 0.1758
final val loss : 3.4913
best val loss : 3.4733
Avg Accuracy  : 12.8906%
epochs       : 300
training time : 05:09
done

```

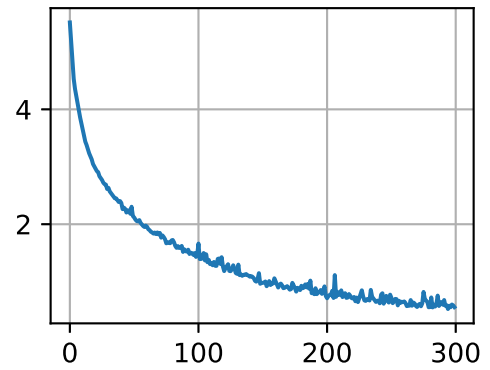
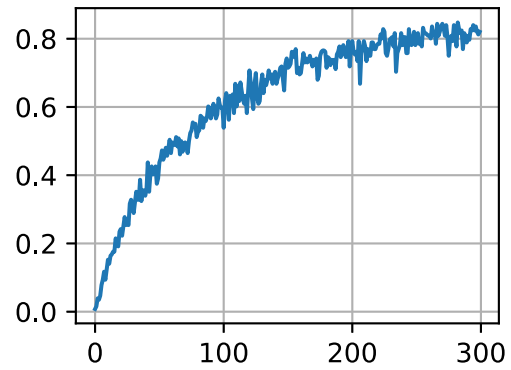
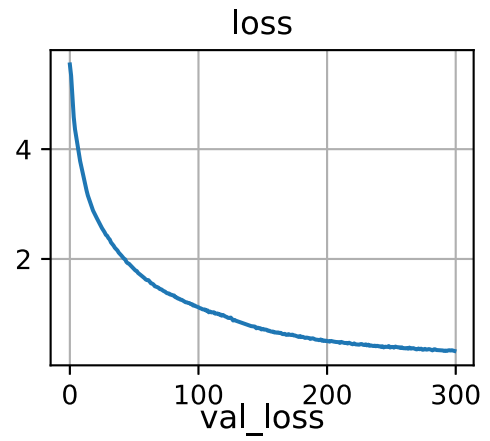
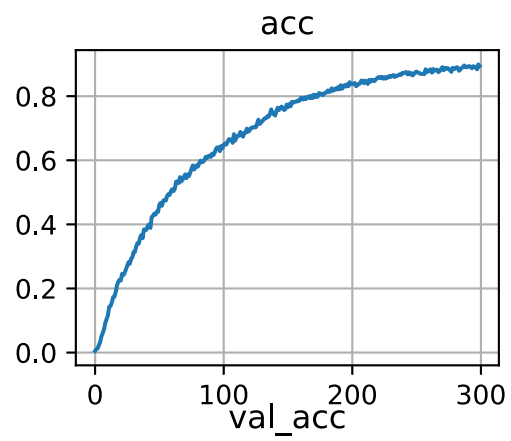
## Model 1 : 2 Conv2D layers

### Model Graph



## Performance

model1 20171023\_142930



***Output***

```

timestamp : 20171023_142930
config    : config/model1/level1-N39W120.json
datafile  : data/level1/N39W120.hgt
divisor   : 16
augments  : 15
epochs    : 300
model     : model1
input shape      : (1200, 1200)
subdivided shape : (256, 75, 75)
normalized shape : (256, 75, 75)
X shape         : (4096, 75, 75, 1)
y shape         : (4096,)
model init
train data      X: (3072, 75, 75, 1) y: (3072, 256)
validation data X: (256, 75, 75, 1) y: (256, 256)
test data       X: (256, 75, 75, 1) y: (256, 256)

```

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 75, 75, 32)	320
dropout_1 (Dropout)	(None, 75, 75, 32)	0
max_pooling2d_1 (MaxPooling2)	(None, 37, 37, 32)	0
conv2d_2 (Conv2D)	(None, 37, 37, 32)	9248
dropout_2 (Dropout)	(None, 37, 37, 32)	0
max_pooling2d_2 (MaxPooling2)	(None, 18, 18, 32)	0
global_average_pooling2d_1 (	(None, 32)	0
dense_1 (Dense)	(None, 256)	8448
Total params: 18,016		
Trainable params: 18,016		
Non-trainable params: 0		

```

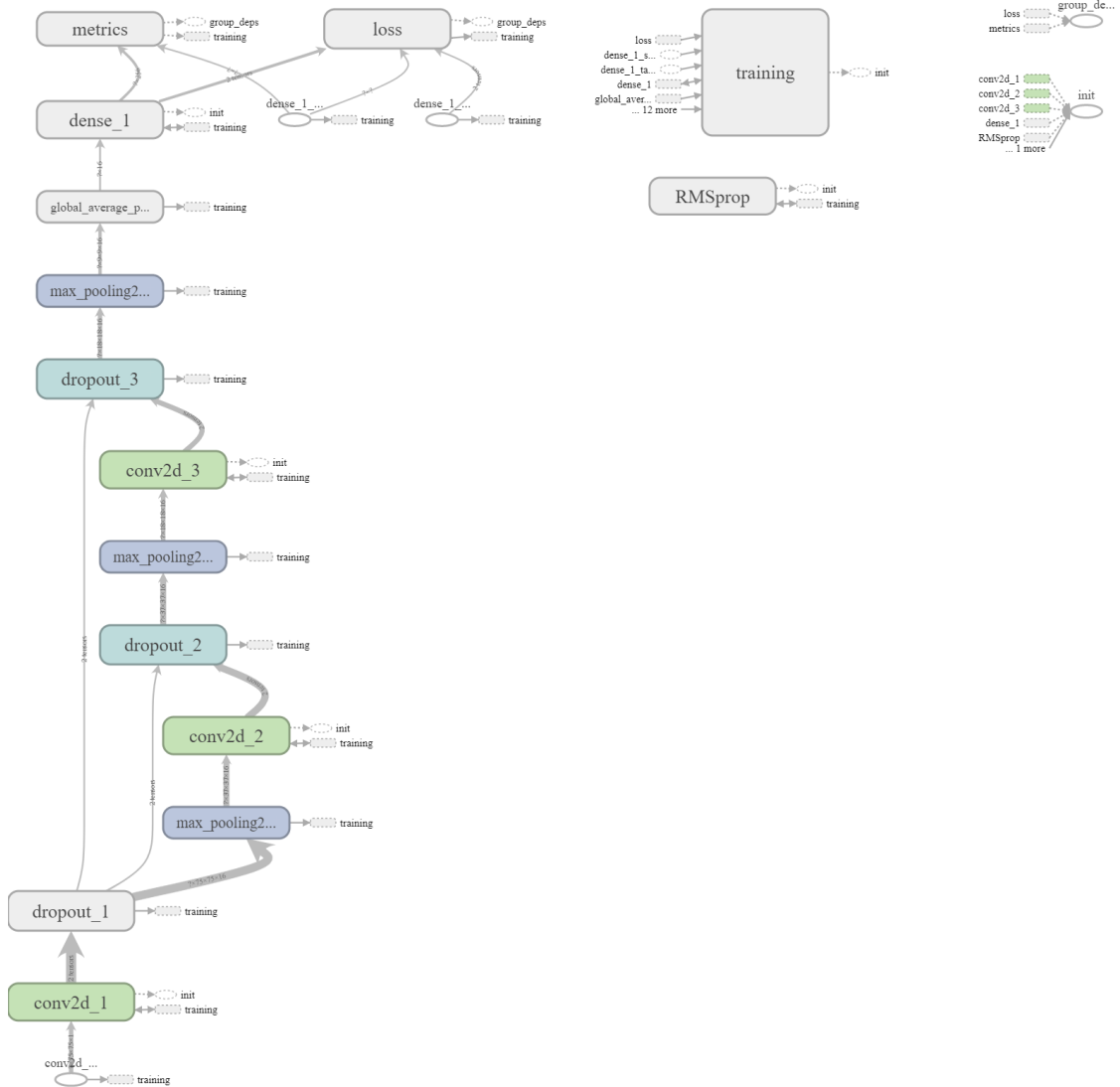
loss : 0.6016498990356922
acc  : 0.84765625
best accuracy : 0.8991
final loss    : 0.3244
best loss     : 0.3235
final val acc : 0.8203
best val acc  : 0.8477
final val loss : 0.5568
best val loss : 0.5235

```

Avg Accuracy : 84.7656%  
epochs : 300  
training time : 11:28

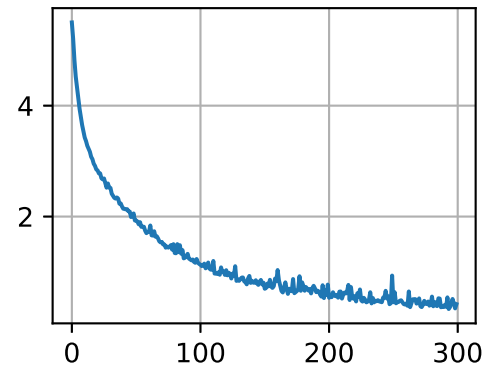
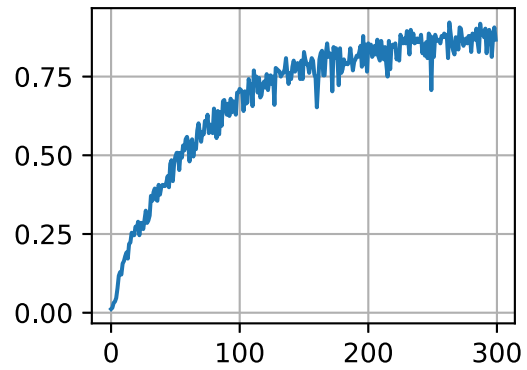
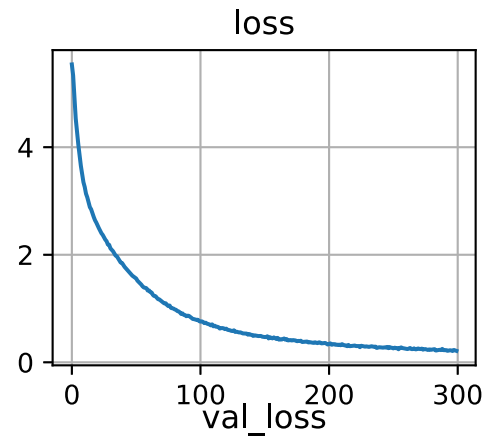
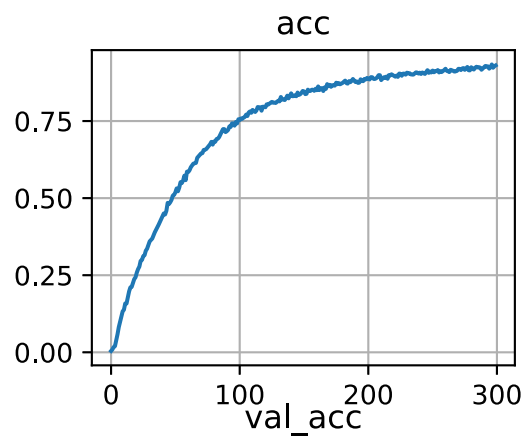
## Model 2 : 3 Conv2D layers, 16 filters, kernel=3

### Model Graph



### Performance

model2 20171023\_144110



***Output***



```

timestamp : 20171023_144110
config    : config/model2/level1-N39W120.json
datafile  : data/level1/N39W120.hgt
divisor   : 16
augments  : 15
epochs    : 300
model     : model2
input shape      : (1200, 1200)
subdivided shape : (256, 75, 75)
normalized shape : (256, 75, 75)
X shape         : (4096, 75, 75, 1)
y shape         : (4096,)
model init
train data      X: (3072, 75, 75, 1) y: (3072, 256)
validation data X: (256, 75, 75, 1) y: (256, 256)
test data       X: (256, 75, 75, 1) y: (256, 256)

```

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 75, 75, 16)	160
dropout_1 (Dropout)	(None, 75, 75, 16)	0
max_pooling2d_1 (MaxPooling2D)	(None, 37, 37, 16)	0
conv2d_2 (Conv2D)	(None, 37, 37, 16)	2320
dropout_2 (Dropout)	(None, 37, 37, 16)	0
max_pooling2d_2 (MaxPooling2D)	(None, 18, 18, 16)	0
conv2d_3 (Conv2D)	(None, 18, 18, 16)	2320
dropout_3 (Dropout)	(None, 18, 18, 16)	0
max_pooling2d_3 (MaxPooling2D)	(None, 9, 9, 16)	0
global_average_pooling2d_1 (GlobalAveragePooling2D)	(None, 16)	0
dense_1 (Dense)	(None, 256)	4352

```

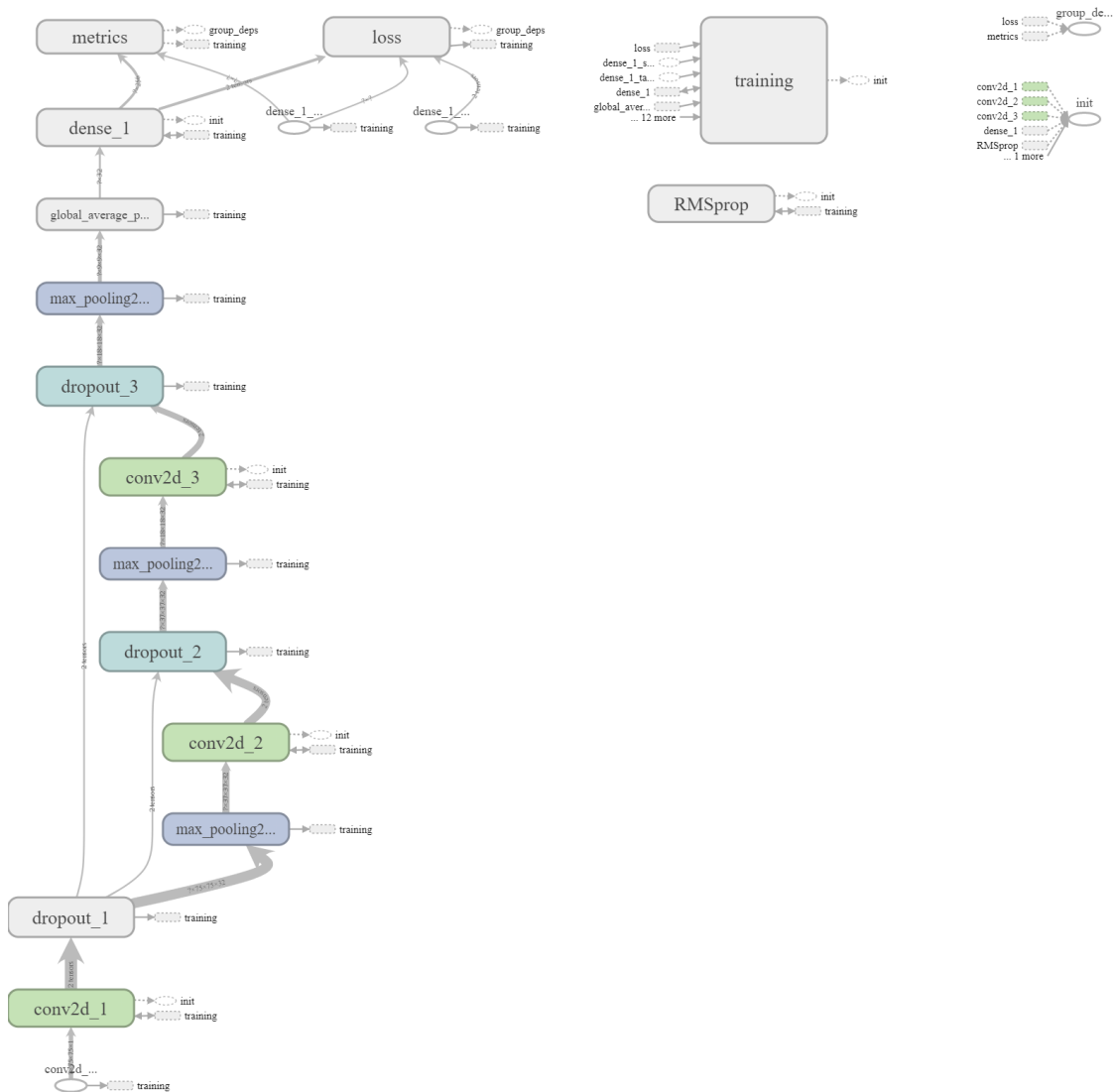
=====
Total params: 9,152
Trainable params: 9,152
Non-trainable params: 0
loss : 0.33772650733590126
acc  : 0.91796875
best accuracy : 0.9336
final loss    : 0.2129
best loss     : 0.2103

```

final val acc : 0.8672  
best val acc : 0.9219  
final val loss : 0.4205  
best val loss : 0.3326  
Avg Accuracy : 91.7969%  
epochs : 300  
training time : 08:12

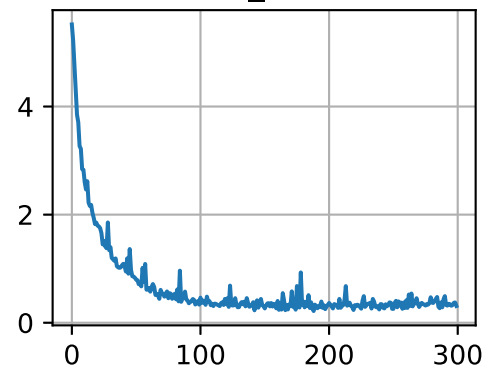
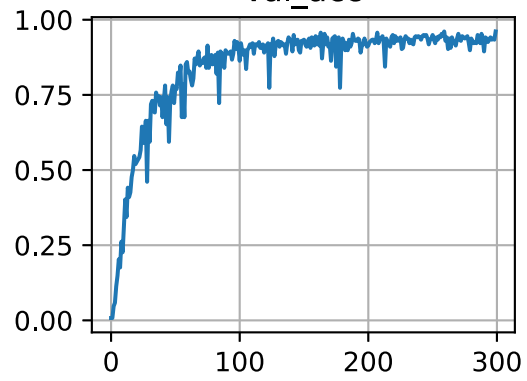
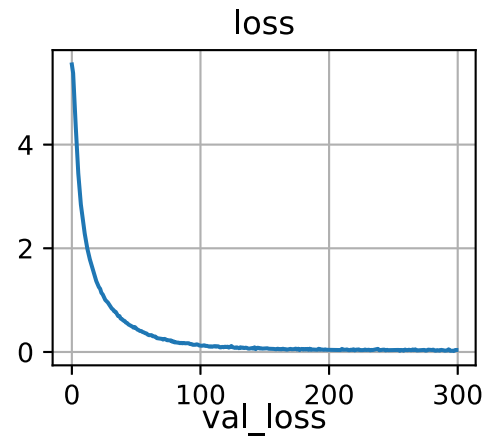
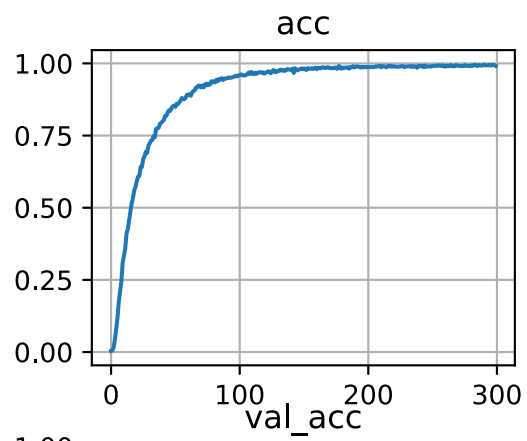
### Model 3 : 3 Conv2D layers, 32 filters, kernel=5

#### Model Graph



#### Performance

model3 20171023\_144934



***Output***

```

timestamp : 20171023_144934
config    : config/model3/level1-N39W120.json
datafile  : data/level1/N39W120.hgt
divisor   : 16
augments  : 15
epochs    : 300
model     : model3
input shape      : (1200, 1200)
subdivided shape : (256, 75, 75)
normalized shape : (256, 75, 75)
X shape         : (4096, 75, 75, 1)
y shape         : (4096,)
model init
train data      X: (3072, 75, 75, 1) y: (3072, 256)
validation data X: (256, 75, 75, 1) y: (256, 256)
test data       X: (256, 75, 75, 1) y: (256, 256)

```

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 75, 75, 32)	832
dropout_1 (Dropout)	(None, 75, 75, 32)	0
max_pooling2d_1 (MaxPooling2)	(None, 37, 37, 32)	0
conv2d_2 (Conv2D)	(None, 37, 37, 32)	25632
dropout_2 (Dropout)	(None, 37, 37, 32)	0
max_pooling2d_2 (MaxPooling2)	(None, 18, 18, 32)	0
conv2d_3 (Conv2D)	(None, 18, 18, 32)	25632
dropout_3 (Dropout)	(None, 18, 18, 32)	0
max_pooling2d_3 (MaxPooling2)	(None, 9, 9, 32)	0
global_average_pooling2d_1 (GlobalAveragePooling2D)	(None, 32)	0
dense_1 (Dense)	(None, 256)	8448

```

=====
Total params: 60,544
Trainable params: 60,544
Non-trainable params: 0

```

```

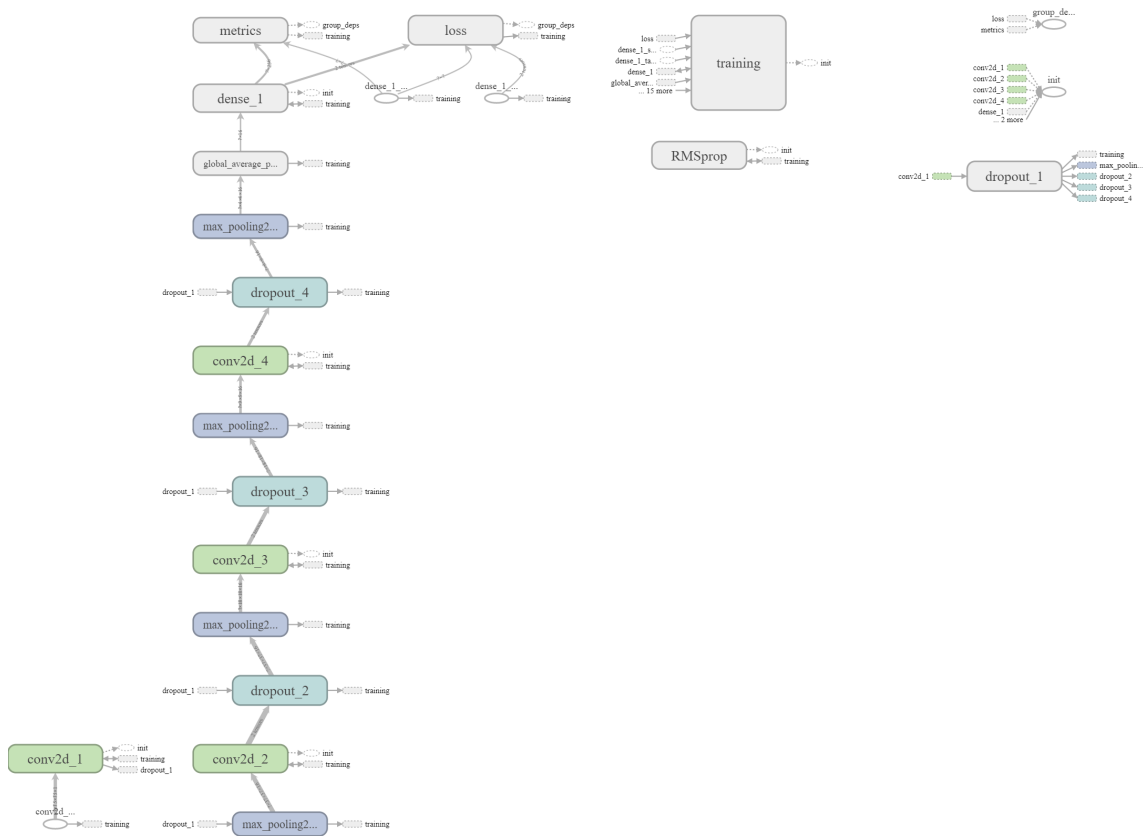
loss : 0.16496535087935627
acc  : 0.94140625
best accuracy : 0.9964
final loss    : 0.0384

```

```
best loss      : 0.0188
final val acc  : 0.9609
best val acc   : 0.9609
final val loss : 0.3100
best val loss  : 0.2164
Avg Accuracy   : 94.1406%
epochs         : 300
training time  : 14:24
```

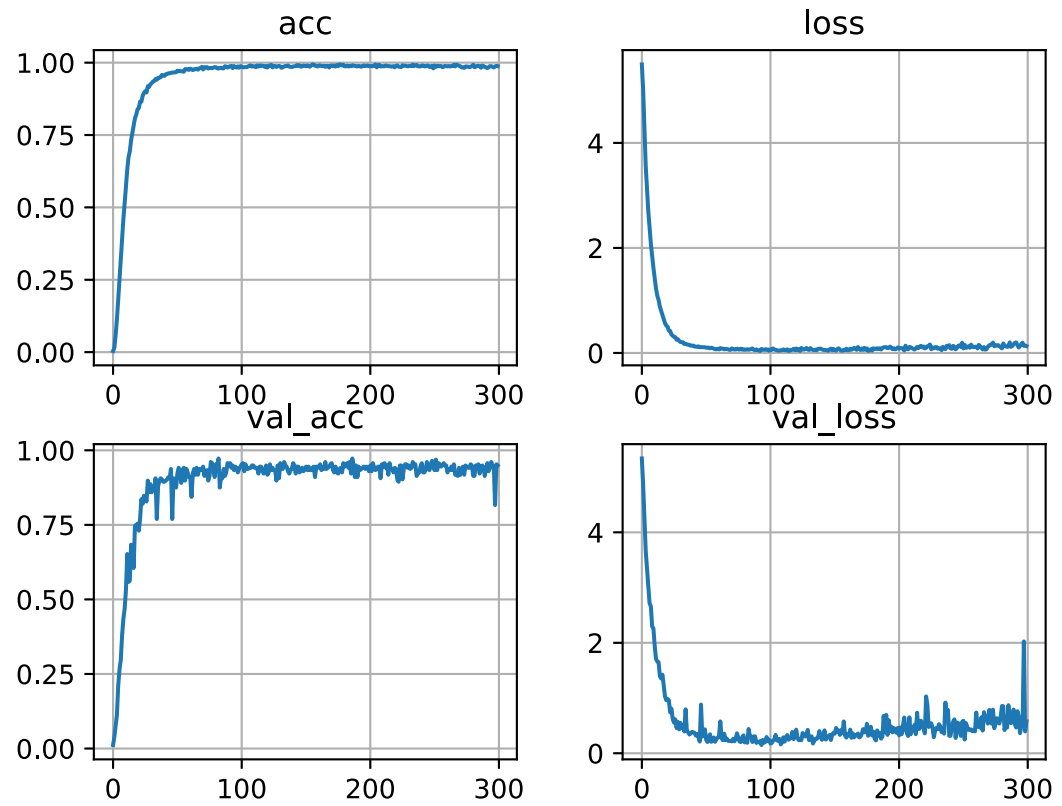
**Model 4 : 4 Conv2D layers, 32 filters, kernel=5**

### Model Graph



## Performance

model4 20171023\_150411



**Output**

```

timestamp : 20171023_150411
config    : config/model4/level1-N39W120.json
datafile  : data/level1/N39W120.hgt
divisor   : 16
augments  : 15
epochs    : 300
model     : model4
input shape      : (1200, 1200)
subdivided shape : (256, 75, 75)
normalized shape : (256, 75, 75)
X shape         : (4096, 75, 75, 1)
y shape         : (4096,)
model init
train data      X: (3072, 75, 75, 1) y: (3072, 256)
validation data X: (256, 75, 75, 1) y: (256, 256)
test data       X: (256, 75, 75, 1) y: (256, 256)

```

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 75, 75, 32)	832
dropout_1 (Dropout)	(None, 75, 75, 32)	0
max_pooling2d_1 (MaxPooling2)	(None, 37, 37, 32)	0
conv2d_2 (Conv2D)	(None, 37, 37, 32)	25632
dropout_2 (Dropout)	(None, 37, 37, 32)	0
max_pooling2d_2 (MaxPooling2)	(None, 18, 18, 32)	0
conv2d_3 (Conv2D)	(None, 18, 18, 32)	25632
dropout_3 (Dropout)	(None, 18, 18, 32)	0
max_pooling2d_3 (MaxPooling2)	(None, 9, 9, 32)	0
conv2d_4 (Conv2D)	(None, 9, 9, 32)	25632
dropout_4 (Dropout)	(None, 9, 9, 32)	0
max_pooling2d_4 (MaxPooling2)	(None, 4, 4, 32)	0
global_average_pooling2d_1 (	(None, 32)	0
dense_1 (Dense)	(None, 256)	8448
Total params: 86,176		
Trainable params: 86,176		

Non-trainable params: 0

---

loss : 0.1833265123423189  
acc : 0.9453125  
best accuracy : 0.9938  
final loss : 0.1322  
best loss : 0.0372  
final val acc : 0.9492  
best val acc : 0.9727  
final val loss : 0.5785  
best val loss : 0.1492  
Avg Accuracy : 94.5312%  
epochs : 300

## IV. Results

### Model Evaluation and Validation

The final model (Model 4) was derived by starting with the Benchmark and successively adding layers with varying hyperparameters. The results of Model 4 seemed too good to be true. The successive loss/accuracy scores did indicate an improvement in each model. My guess is that the dataset is too limited. It has 256 base images, with augmentation increasing that to 4096 images total. The lower resolution level 1 (1200x1200) data files, divided into 16x16 subimages, resulted in each image being 75x75 pixels. The augmented images had relatively small modification hyperparameters so those images are close to the original.

The change from 16 filters, kernel=3 in model 2, to 32 filters, kernel=5 in models 3 and 4 made a significant difference. Both models 3 and 4 converged very quickly and did not need nearly as many training epochs as were allocated.

The measured loss and accuracy was best in model 3. Model 4 regressed a bit. Although it is not shown here, the models were run on both the N39W120 dataset and the N37W098 datasets. Both datasets reflected the regression in model 4 from model 3. The N37W098 results are in the 'results' directory. (The .txt file in each timestamped results folder shows which dataset was used.)

A possible explanation of the regression is that, although in all cases there was almost no divergence between the training curves and the validation curves, the model 4 validation loss showed a slight upward slope that was not present in the model 3 curves. This could indicate that model 4 was beginning to overfit.

### Results from N39W120 (mountainous terrain)

- Benchmark
  - loss : 3.434
  - acc : 0.128
- Model 1



- loss : 0.601
- acc : 0.847
- Model 2
  - loss : 0.337
  - acc : 0.917
- Model 3
  - loss : **0.164**
  - acc : **0.941**
- Model 4
  - loss : **0.183**
  - acc : **0.945**

### Results from N37098 (flat terrain)

- Model 1
  - loss : 1.115
  - acc : 0.73
- Model 2
  - loss : 0.538
  - acc : 0.871
- Model 3
  - loss : **0.110**
  - acc : **0.972**
- Model 4
  - loss : **0.429**
  - acc : **0.937**

## Justification

The best solution was with model 3. It was much better than the benchmark. Much of this improvement could be due to the simplicity of the input data. It appears the input data happens to be very friendly to a simple convolutional neural network.


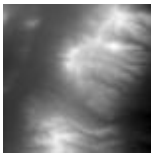
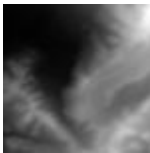


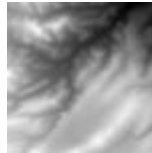

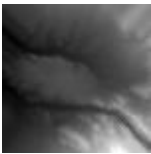
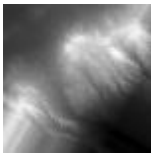


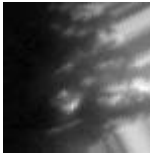
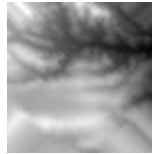

One question to be answered is whether the predictions on the test data are real. Some visualization is needed to confirm that. Using model 3, the code was modified to store the base images before subdivision, and the test image set. Then the model was re-run and some examples shown here of test images, both those where the prediction matched the label, and those that did not match. Those images are shown in the next section.

## V. Conclusion

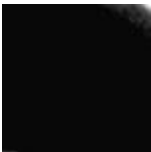
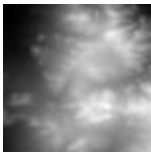


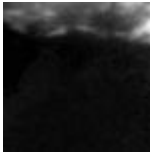





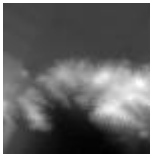

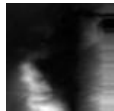
### Free-Form Visualization

The images in the following two tables show examples from a run of Model 1 where some test images were predicted correctly and some images were predicted incorrectly. The 'Input' is the original image before augmentation. The 'Predicted' is one of the augmented images matching the predicted label. These images are stored in the github project under directory 'images/model1'.

**Example Base vs Test images for Model 3 where the input label and predicted label match**

INPUT LABEL						
1	51	101	131	191	201	255
						
1	51	101	131	191	201	255
PREDICTED LABEL						
1	51	101	131	191	201	255
						

**Example Base vs Test images for Model 3 where the input label and predicted label did not match**

INPUT LABEL						
24	69	123	133	174	177	188
						
PREDICTED LABEL						
221	164	109	219	195	87	212
						

**Reflection**

I think my original proposal had a bit too much scope. I followed it, but I found I did not have time to do everything I wanted to do. These results would just be a first step to creating anything resembling a useful system for determining position. While the recognition of images worked well, the simple division in NxN sections is not good enough. A real system would need to take overlapping sections into account. It would need a process that detected subsections within a larger image.

At this point I do not know enough about the innards of the various neural net layers and how they could be used to improve a model. It just seems like a combinatorial problem, trying things until something works. That is clearly not a good approach. I did learn a lot about using Numpy to process data and refined my Python skills a bit. I think at this point I could serve as an implementer of a neural net processing pipeline if someone with more theoretical knowledge specified the model itself.

### **data preprocessing**

This was the majority of coding time spent. I had to figure out how to read the SRTM data, convert it to numpy arrays, write it as images and format it as tensors for the model. Much of the time was spent refactoring the preprocessing code to provide a clean pipeline. I learned a lot about using numpy efficiently. My first cut of code used explicit loops (I'm a C programmer), which I then refactored to use slicing and numpy array functions. My lib.srtm module probably duplicated existing functions in numpy, scipy and PIL but doing it myself helped me improve my skillset.

### **model processing**

While the models themselves were very simple and did not require much coding, I spent a lot of time constructing and refactoring the processing pipeline to be general enough to run different models without code change, using the .json configuration file.

### **directory structure**

I did a lot of refactoring of the project directory structure, both for the data preprocessing and saving the results. I ended up with a lot of directories but I think it is well organized.

### **Numpy, Tensorboard, Keras, Google Compute Engine**

I learned a lot about using Python and Numpy. I used Tensorboard a lot during development. I just touched the surface on Keras and designing models in general. I learned how to use Google Compute Engine, and I found it much easier to use than the Amazon alternative. Unfortunately the GPU support is pretty slow and expensive.

### **Improvement**

The application could be improved by segmenting the individual .hgt files into smaller subsegments, using a system capable of handling the 3600x3600 resolution, along with training the model over a collection of the .hgt files instead of one at a time. Smaller subsegments with higher resolution would make the position resolution more precise. The 1200x1200 resolution files might not provide enough differentiation between similar images. Grouping multiple .hgt files with a proper labeling convention a much larger coverage area could be trained in one pass. The current approach of one .hgt at a time is a limiting factor.

