Udacity

Machine Learning Nanodegree Capstone Project

Ronan Casey
Nov. 20th 2018

Using Convolutional Neural Networks to Identify Icebergs in Dual-Polarization C-Band Radar Images

---

Definition

Project Overview

Sea navigation in The North Atlantic Ocean presents many obstacles to vessels contending with this harsh environment. One of those obstacles takes the form of icebergs. Icebergs are large pieces of freshwater ice which have broken off a glacier or ice shelf and are now floating freely in open water. One mode of identifying icebergs in remote locations relies on the analysis of radar images from an orbiting satellite. The radar images result from sending out radio waves in both vertical and horizontal wave orientation (Dual-Polarization), which provide a more detailed image of objects detected by the radar.

Using the data collected from locations of interest, this project uses current methods of computer vision deep learning processes in the form of Convolutional Neural Networks and Transfer Learning to analyse and classify the objects found in the images and return a probability of whether the object detected is an iceberg or a ship (vessel). This project is inspired by a Kaggle Competition titled “Statoil/C-CORE Iceberg Classifier Challenge” and uses the same radar information data set.

Problem Statement

Considering the time sensitive nature of the identification task, a fast reliable classifier is required for this problem. Once the supplied data has been tidied and pre-processed, it will be ready for training a Convolutional Neural Network (CNN). The CNN must be designed and customized for this specific task. A Transfer Learning approach will allow for the use of a pre-trained CNN to be utilized where the iceberg data is introduced at the input layer and the corresponding identification labels are attached at the output layer with the node values and weights of the the hidden layers being updated with each pass of the training data. Transfer Learning has proven to be a highly effective versatile tool for image

classification tasks. With enough training data, a CNN can be expected to produce similar results to a human observer with a negligible amount of error.

Metrics

This project will be evaluated using a log loss function. The log loss function uses the values of each prediction, where 1 is associated with an iceberg and 0 is associated with a ship. This combined with the prediction probability, ranging from 0.0 to 1.0 is used to measure the performance of the model. The greater the amount of correct predictions and the higher the probability attributed to those predictions will result in a lower log loss value. Models which achieve a lower log loss value are considered to be better performing. After training, the CNN model will analyse previously unseen test data and compare the predictions with the test set labels. The results will be calculated using the log loss function where values closer to zero are optimal.

The formula for the log loss function is as follows:

$$logLoss=-1N\sum Ni=1(yi(logpi)+(1-yi)log(1-pi))$$

Analysis

Data Exploration

In this project, the dataset used is the same one provided with the Kaggle Competition. The data consists of two json formatted files within which a list of images have been split into training and testing sets. The testing set does not contain a class identifier vector, as this set is used only to generate predictions. The test predictions are then scored via the Kaggle submission process. Both sets include the following fields for each image:

- **Id –** the unique id of the image
- **band_1, band_2 –** the flattened image data. Each band consists of 75x75 pixel values in a list, where each list contains a total of 5625 elements. The value of each element is represented as a floating point number on a dB scale. Band 1 and Band 2 are signals characterized by radar backscatter produced from different polarizations at a particular incidence angle. The polarizations correspond to HH (transmit/receive horizontally) and HV (transmit horizontally and receive vertically).
- **Inc_angle –** the incidence angle at which the image was captured. Some of the training set values for this field appear as “na”. This must be dealt with appropriately before fitting the model.

Unique to the training set is the following field:

- **is_iceberg –** the target variable, set to 1 if an iceberg is present, and 0 if a ship is present.

Exploratory Visualization

The image data from a radar source is known as “backscatter”. The data for this project contains two bands of data for each image id. Since the data is in a dB scale, it is possible to generate a low resolution 3D model using the backscatter from band 1. Fig. 1a (below left) shows an example of an iceberg from the training data set. The shape is characteristically irregular with numerous peaks and valleys and smooth contours where the iceberg meets the water.

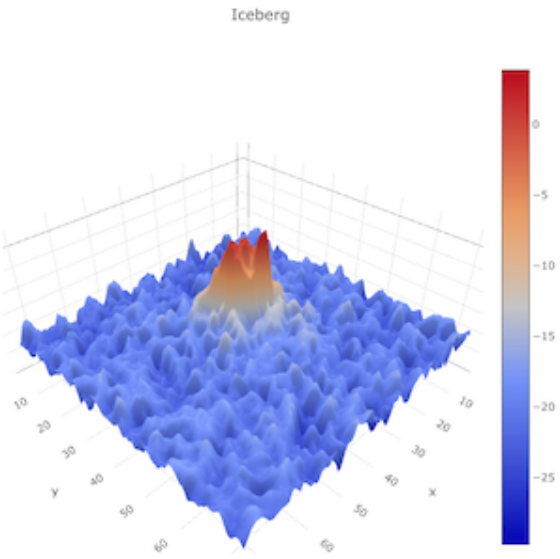


Fig. 1a

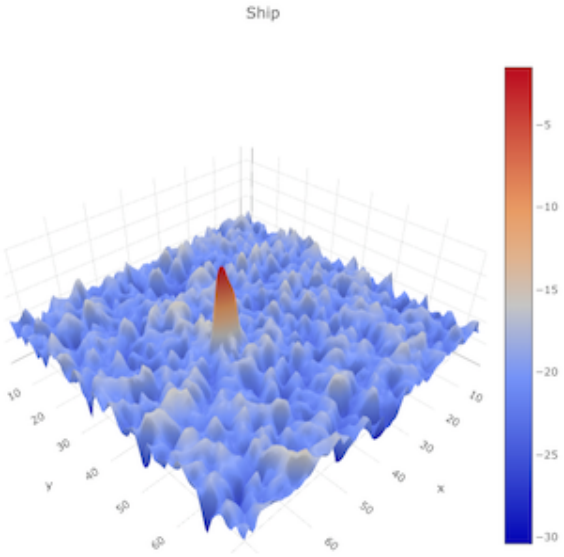


Fig. 1b

Fig. 1b (above right) shows a ship. Here we can see the shape is much more uniform and resembles a steep contoured pillar protruding from the water. These distinct characteristics will provide the largest degree of feature differentiation during the CNN training process.

Algorithms and Techniques

Given that the radar data can be arranged as a 2D array (similar to an image), a Convolutional Neural Network (CNN) will be well suited to the classification task presented. The general structure of a CNN consists of the following components:

Input – For the purposes of image classification, this is a 2D image. Where colours are present, the image will consist of 3 channels (RGB). Because our iceberg data only consists

of two channels (Band 1, Band 2), it was necessary to create a third channel, Band 3. Band 3 is made up of the average value between Band 1 and Band 2 for each pixel.

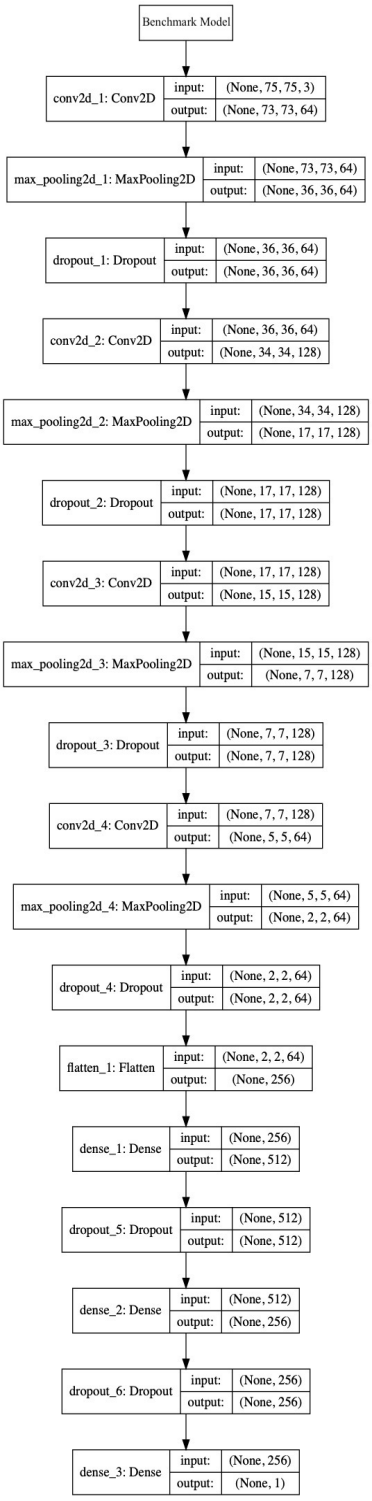
Convolutional Layer – The operation performed in this layer uses a selected number of filters to scan through the image in incremental steps. Among various parameters which can be set, kernel size and stride size of the filter window can be changed according to the input. Convolutional layers are where the features unique to the input stage are generated before being passed on to the next layer in the chain. Each filter is defined by the co-efficient and weight, where the values are generated during a forward pass and updated and fine-tuned by backpropagation. Convolutional layers are typically followed by an activation function. The activation function selected is integral to the performance of the model. The image classification task favours the use of non-linear activation functions like Rectified Linear Unit (relu) activation.

Pooling Layer – Pooling layers typically follow convolutional layers. These layers combine clusters of neurons from the associated convolutional layer and reduce each cluster to a single neuron for the next layer to process. The degree of dimensionality reduction is determined by the pool size. Max pooling takes the maximum value in the cluster, whereas average pooling takes the average value for the cluster before passing this value to the input stage of the next layer.

Fully Connected Layer – Neurons in a fully connected layer have full connections to all activations in the previous layer. As the final layer in a CNN, this layer will compute the class scores attributable to the input data. Each of the two possible objects (iceberg, ship) correspond to a class score presented as the predicted probability of each object being in the image.

One of the requirements for a CNN to perform well is that it will need a lot of training data in order to fit the model accurately. This can present a problem where the training sets are limited in size. One way to address the lack of training data is use image augmentation generators to shift angles and zoom settings, giving the CNN more similar, though not identical data in the process. Another method to employ is the integration of a pre-trained network. The process of using a pre-trained model like this is known as “Transfer Learning”, where the network architecture of hidden layers and parameters come pre-tuned, and need only be adjusted in order to fit our problem.

Benchmark



.

Fig. 2

A regular CNN will be constructed for benchmark purposes. The method here is to design a CNN which is trained exclusively on the iceberg training data, where all weights are updated from an initialised state.

The benchmark architecture can be seen in Fig. 2 (left)

When compiling the model, the loss function chosen is binary cross-entropy, the optimizer is Adam with default hyper-parameter values, and the metric is Accuracy.

The model is set to stop training if the validation loss has not decreased in the previous 10 epochs. Models often benefit from reducing the learning rate once learning stagnates. The benchmark model will reduce the learning rate by a factor of 0.1 if the validation loss has not been reduced in the previous 7 epochs.

The model was trained with a limit of 50 epochs and a validation split of 0.25 (401 observations), using the remaining 75% (1203 observations) of the data for training the CNN.

Using the best weights from the trained model, the test labels were predicted and produced the following result on Kaggle:

0.2284

The resulting log loss score for the test predictions is better than expected. Given the limited amount of training data and the relatively basic network architecture, the benchmark model is reasonably well tuned to the data.

Methodology

Data Pre-processing

Missing Values

The supplied data is presented in json format. According to the Kaggle competition notes, the variable “inc_angle” has some missing values for the training set. The missing values are shown with a string value “na”. To begin, the whole “inc_angle” column is converted to numeric values. This step will replace all “na” values with null values using the following pandas function:

```
train['inc_angle'] = pd.to_numeric(train['inc_angle'], errors='coerce')
```

The indices relating to missing values were found at the following positions:

```
In [47]: # inspect rows missing inc_angle values
         train.loc[train['inc_angle'].isnull()].index

Out[47]: Int64Index([ 101,  126,  142,  145,  176,  179,  191,  278,  286,  342,
                      ...
                      1594, 1595, 1596, 1597, 1598, 1599, 1600, 1601, 1602, 1603],
                     dtype='int64', length=133)
```

133 observations contain missing values for “inc_angle”. Machine learning processes are sensitive to incomplete data, so these values will have to be estimated using the data which is present for the incidence angle of the other observations. One tool which can be called upon for this task is “fancyimpute”. Using the K Nearest Neighbour algorithm within fancyimpute, the missing values are generated according to feature similarities of complete observations. All null values were successfully estimated by the algorithm:

```
         # confirm no values are missing
         train.loc[train['inc_angle'].isnull()].index

Out[52]: Int64Index([], dtype='int64')
```

Missing Channel

The input stage of a CNN for image classification will be expecting three channels of data corresponding to red, green and blue in a standard colour image. Our data only contains values for two channels (band_1, band_2). A third channel must be created to allow the pre-trained network to function optimally. Seeing as the dB SPL scale is logarithmic, the third channel will be the sum of the values in the first two channels per pixel. The band data for

each observation is presented as a flattened 1D array, reshaping this data to a 2D 75x75 array will be required as per image processing specifications:

```
def get_scaled_imgs(df):
    imgs = []

    for i, row in df.iterrows():
        #make 75x75 image
        band_1 = np.array(row['band_1']).reshape(75, 75)
        band_2 = np.array(row['band_2']).reshape(75, 75)
        band_3 = band_1 + band_2 # plus since log(x*y) = log(x) + log(y)

        # Rescale
        a = (band_1 - band_1.mean()) / (band_1.max() - band_1.min())
        b = (band_2 - band_2.mean()) / (band_2.max() - band_2.min())
        c = (band_3 - band_3.mean()) / (band_3.max() - band_3.min())

        imgs.append(np.dstack((a, b, c)))

    return np.array(imgs)
```

Limited Training Data

To address the limited amount of training data available, I was able to generate extra augmented data from the existing training data. Using the cv2 package for python, the original images were flipped both vertically and horizontally. The resulting training set contains 3 times as many images fort the CNN to train on. The function to create these extra images is below.

```
def get_more_images(imgs):

    more_images = []
    vert_flip_imgs = []
    hori_flip_imgs = []

    for i in range(0,imgs.shape[0]):
        a=imgs[i,:,:,0]
        b=imgs[i,:,:,1]
        c=imgs[i,:,:,2]

        av=cv2.flip(a,1)
        ah=cv2.flip(a,0)
        bv=cv2.flip(b,1)
        bh=cv2.flip(b,0)
        cv=cv2.flip(c,1)
        ch=cv2.flip(c,0)

        vert_flip_imgs.append(np.dstack((av, bv, cv)))
        hori_flip_imgs.append(np.dstack((ah, bh, ch)))

    v = np.array(vert_flip_imgs)
    h = np.array(hori_flip_imgs)

    more_images = np.concatenate((imgs,v,h))

    return more_images
```

Implementation

Environment

After initially attempting to run the algorithms on my modest home laptop, it quickly became clear that more resources were going to be necessary to allow for multiple model assessments. Since I had some credit remaining from a previous lesson, I was able to set up

an AWS instance with a few minor caveats which will be discussed. Below is an overview of the environment configuration.

Virtual Machine:	Deep Learning AMI Ubuntu Linux
Instance:	p2.xlarge
Environment:	Jupyter Notebook
Language:	Python 3.6 (with numpy, pandas, matplotlib and cv2)
CNN Library:	Keras (Tensorflow)

Model Selection

The model selected for the purposes of transfer learning is the VGG16 model with weights corresponding to the "imagenet" dataset. The model is edited to accept the same image dimensions as our dataset (75, 75, 3). In order to benefit from VGG16s pre-tuned weights, the network is interrupted following the final convolutional block, where the output of the same block is passed through a global max pooling layer before being processed by 3 fully connected layers with varying density and dropout regularity. Each of these fully connected layers have their activation function set to the non-linear relu activation. The final layer of the network is a fully connected layer of density 1 and an activation step sigmoid, as our classification problem is binary. A flowchart representing the final transfer learning model can be seen in Fig. 3 (below)

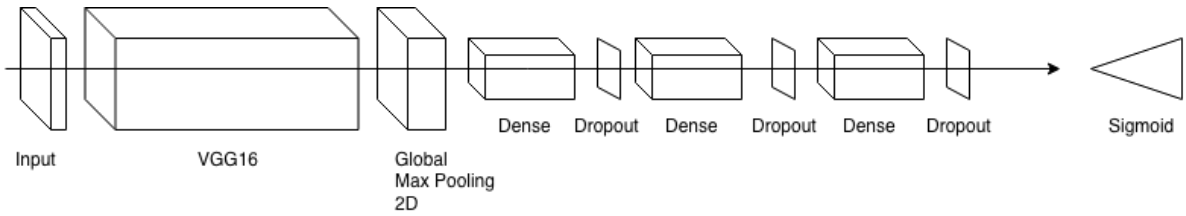


Fig. 3

Complications

When using the AWS instance, the system would not allow me to install a number of packages which I had used locally to explore and fix some of the data. As a result, I made a new dataset with the imputed missing values and used that dataset on the virtual machine.

As json outputs data in string format, it was necessary to convert these values to numeric where appropriate. An example of this process can be seen below.

```
# Convert inc_angle to numeric
train['inc_angle'] = pd.to_numeric(train['inc_angle'], errors='coerce')
test['inc_angle'] = pd.to_numeric(test['inc_angle'], errors='coerce')
```

Refinement

Initially training the benchmark model did not take very long. The same could not be said for the transfer learning model, however. After experimenting with various architecture iterations, the results were far from satisfactory. Initially, I hadn't taken measures to increase the training data by augmentation. After including the augmented data, the result improved slightly with a longer training time being the cost.

I experimented with various pooling layers between VGG16 and fully connected layers. My finding was that "Global Max Pooling" lead to faster loss reduction per epoch for both training and validation sets. Other pooling options tried were "Global Average Pooling" and "Flatten".

Many combinations of fully connected layers were assessed. I found 3 fully connected layers -with depth 516, 256 and 128 respectively- to be optimal for this dataset. The dropout regularity was also repeatedly assessed. I found the final settings –with proportions 0.5, 0.2 and 0.5 respectively- were associated with quicker convergence and smaller cross validation loss values.

As an optimizer, I found that Stochastic Gradient Descent (SGD) was the best fit for this dataset. SGD consistently produced lower loss convergence scores compared to the other optimizer I tried, "Adam". The transfer learning model was set to train on all layers, and not restricted to training on the fully connected layers only. Attempts to train on the fully connected layers only prevented the model from achieving scores comparable to the benchmark model.

The final transfer learning model settings can be seen below.

```
# Connect transfer learning network
def tlearnModel():

    base_model = VGG16(weights='imagenet', include_top=False,
                       input_shape=Xtrain.shape[1:], classes=1)

    x = base_model.get_layer('block5_pool').output

    x = GlobalMaxPooling2D()(x)
    #x = GlobalAveragePooling2D()(x)
    #x = Flatten()(x)
    x = Dense(512, activation='relu')(x)
    x = Dropout(0.5)(x)
    x = Dense(256, activation='relu')(x)
    x = Dropout(0.2)(x)
    x = Dense(128, activation='relu')(x)
    x = Dropout(0.5)(x)

    predictions = Dense(1, activation='sigmoid')(x)

    model = Model(inputs=base_model.input, outputs=predictions)

    adam = Adam(lr=1e-3, decay=0.0)
    sgd = SGD(lr=1e-3, decay=1e-6, momentum=0.9, nesterov=True)
    model.compile(loss='binary_crossentropy', optimizer=sgd, metrics=['accuracy'])

    return model
```

Even though all of these choices improved the test results, they were still not approaching a score where you could say definitively that there was a marked improvement. The biggest contributing factor to finally getting a satisfactory score was setting up a more sophisticated cross validation split. The method used for this project is Stratified K Fold, where a proportionally consistent number from each class is put into k groups. Each of these groups is then used in turn as a validation set, where the remaining groups represent the training set. As the model is trained in k separate instances, the final training scores, validation scores and test predictions are averaged out across all training instances.

```
# Create train/test indices to split data in train/test sets
n_splits = 5
kfold = StratifiedKFold(n_splits=n_splits, shuffle=True, random_state=seed)
```

Larger batch sizes definitely cause the model training to converge quicker, but I found that smaller sizes helped the training/validation loss metric to converge at a lower score. This is associated with a more robust model. I eventually settled for a batch size of 8, which lead to lower loss convergence at the expense of longer training times.

Results

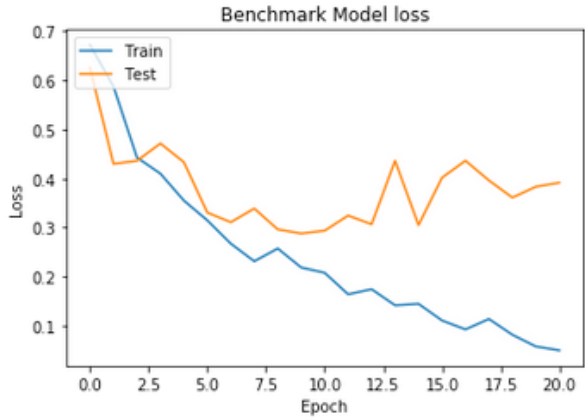
Model Evaluation and Validation

The solution model has and early stopping limit of 5, meaning that the training session will terminate if the minimum validation loss score has not decreased in the 5 previous epochs. The learning rate of the optimizer is set to be multiplied by 0.1 if the minimum validation loss has not decreased in the previous 3 epochs.

```
batch_size = 8
earlyStopping = EarlyStopping(monitor='val_loss', patience=5, verbose=0, mode='min')
mcp_save = ModelCheckpoint('.tlearn_mdl_wts.hdf5', save_best_only=True, monitor='val_loss', mode='min')
reduce_lr_loss = ReduceLROnPlateau(monitor='val_loss', factor=0.1, patience=3, verbose=1, epsilon=1e-4, mode='min')
```

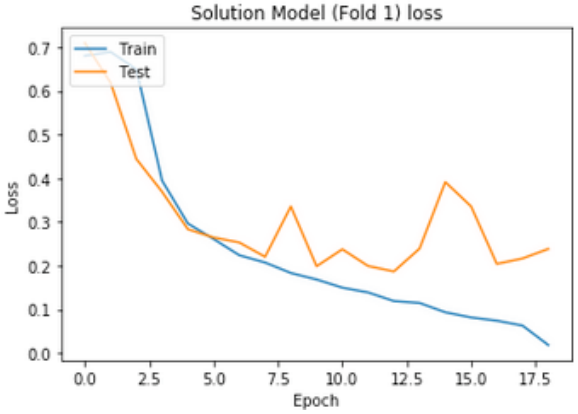
In this section we will compare the evaluation performance of the benchmark model versus the transfer learning model on the training set available to each.

Evaluation metrics for benchmark model trained on unaltered training set with a regular validation split of 25%



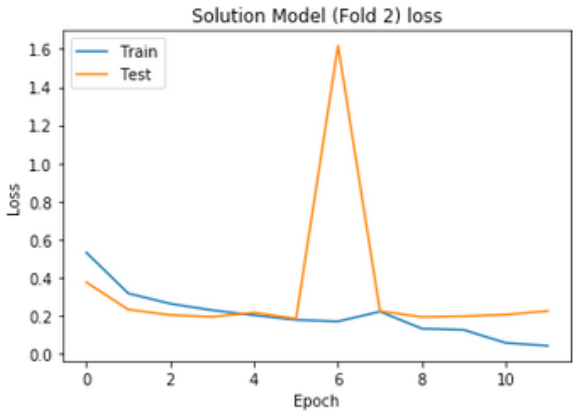

```
Train loss: 0.19445121436
Train accuracy: 0.934538653367
```

Evaluation metrics for transfer learning model trained on augmented and increased size training set with k=5 Stratified K-Fold Cross Validation.



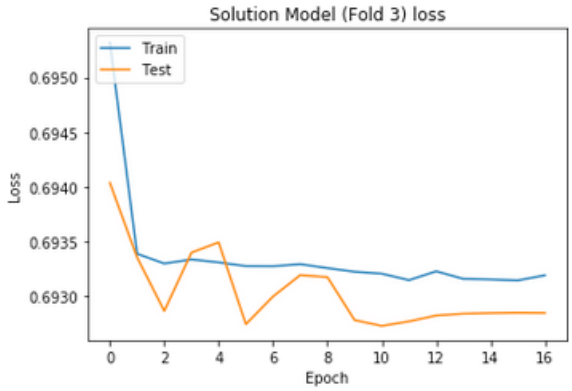

```
Fold 1

Train loss: 0.0509221777129
Train accuracy: 0.982073265783
```



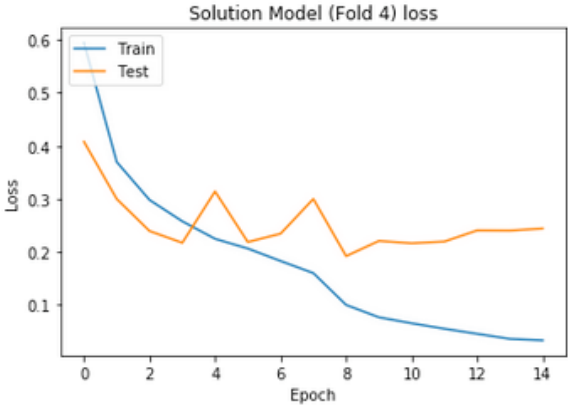

```
Fold 2

Train loss: 0.114635412338
Train accuracy: 0.955572876072
```



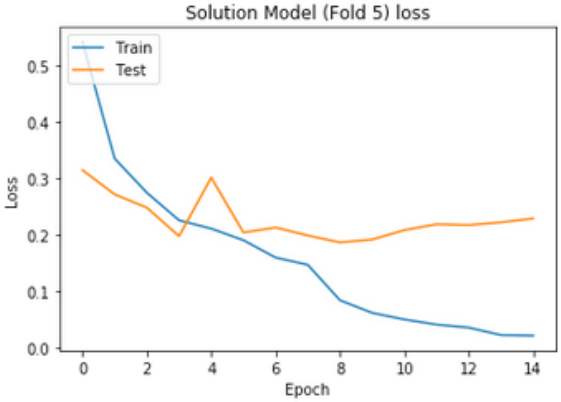
Fold 3

Train loss: 0.129464982572
Train accuracy: 0.949337490257



Fold 4

Train loss: 0.133137071988
Train accuracy: 0.946233766234



Fold 5

Train loss: 0.130587762279
Train accuracy: 0.948325110438

Average evaluation metrics when accounting for all folds.

Train loss: 0.111749481
Train accuracy: 0.956308502

Comparing the evaluation results alone, it is clear to see that we have made a big improvement overall, especially in terms of the loss metric where an improvement of 0.082701733 has been made.

Justification

Ultimately, the most important metric by which to measure the quality of the solution model is the log loss value returned from the Kaggle submission.

Kaggle Benchmark Model Log Loss Score:

0.2284

Kaggle Transfer Learning Solution Model Log Loss Score:

0.1773

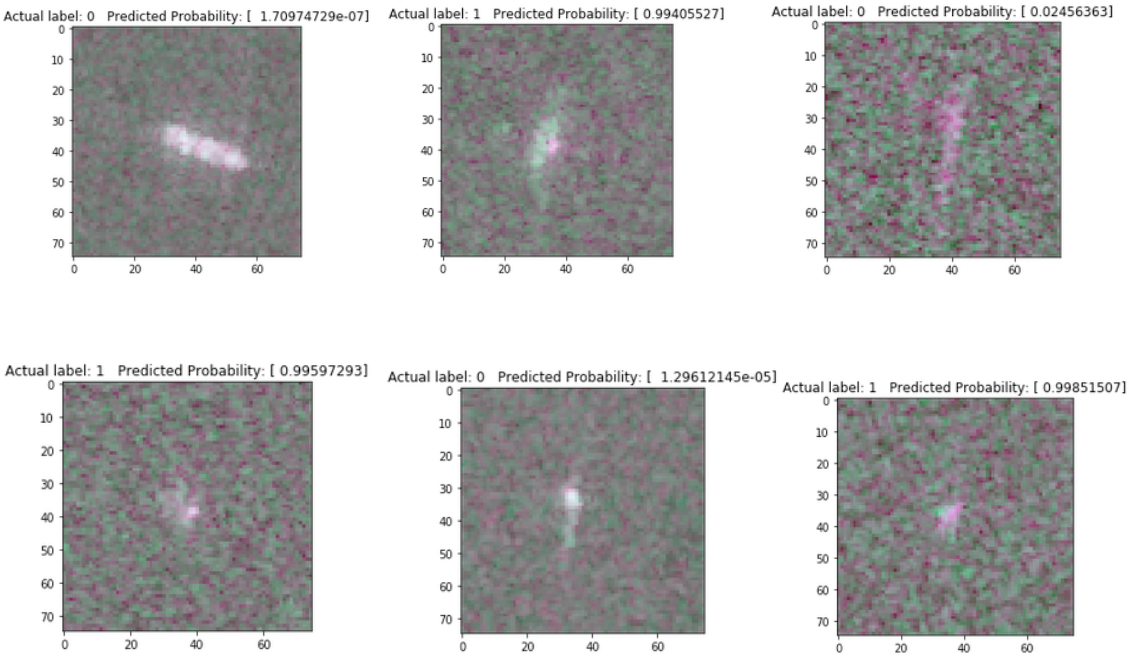
We have achieved a log loss score improvement of `0.0511`

While this score may not be considered enough to have solved the problem, it is certainly a significant improvement compared with the benchmark.

Conclusion

Free-Form Visualization

The following examples show what the model sees when making a prediction. Even though we can not always clearly see whether or not the object is either an iceberg (1) or a ship (0), the model is incredibly accurate at making the correct classification in the majority of cases.



Reflection

The problem addressed in this project is derived from a Kaggle competition where the objective is to successfully identify icebergs from a dataset containing icebergs and ships. This project trained a CNN by way of transfer learning from the VGG16 “imagenet” model. The radar images were prepared for the input stage first by adding a third channel of data, and secondly by reshaping and rescaling. The training set was increased in size by way of vertical and horizontal flipping of the images. Stratified K-Fold Cross Validation was implemented to train/validate on the entire training set. The combination of increasing the training set and performing Stratified K-Fold Cross Validation contributed most to the improvements observed. The results obtained from this model outperformed the benchmark by a significant amount when submitted on the Kaggle platform.

Improvement

Smaller optimizer learning rates and larger cross validation fold sizes seem to be associated with more robust model fitting. With additional time and adequate hardware resources, more experimentation could be made addressing these parameters to find a more robust model. The learning rate used in the solution is 1e-3. Using a smaller rate may improve performance at the expense of training time. The number of stratified k-fold splits in this project is set to 5. I did not experiment with larger split sizes due to time constraints, but noticed a performance improvement form the default value of 3. Increasing this value further may tune the layer weights more precisely.