Machine Learning Engineer Nanodegree

Capstone Project: Dogs vs Cats Redux

Michael Low

March 10th, 2017

Definition

Project Overview

I have chosen to work on the “Dogs vs Cats Redux” Kaggle competition (Kaggle, no date), which is the challenge of automatically distinguishing photos of dogs from photos of cats.

This has been a task that is traditionally very easy for humans, but difficult for computers due to large variety of shapes, breeds, colours, photo composition, lighting and so on in the photos. Ten years ago, it was used as a CAPTCHA challenge (Microsoft, 2007) to distinguish human users of a system from computers. In 2008, techniques in computer vision advanced to sufficiently attack the CAPTCHA with 82.7% accuracy with a SVM classifier (Golle, 2008) making it no longer viable. Modern computer vision techniques using convolutional neural networks should be able to improve on this substantially further still.

I think this is an exciting challenge to work on as, although it may have limited use itself, it encapsulates the fundamental techniques needed to solve a wide range of computer vision problems into a simple problem domain. Consequently, the methods used to achieve high accuracy on the Dogs vs Cats problem can then be applied to a wide range of today's computer vision challenges.

Problem Statement

The problem to be solved is to automatically identify whether a given jpg image is of a cat or a dog, along with a probability reflecting the confidence in that prediction.

The intended way to solve this problem is by using Convolutional Neural Networks (CNNs) and deep learning techniques. The basic steps necessary are:

- Split the training data into training data and validation data sets.
- Construct a multi-layered convolutional neural network.
- Train the network using the Kaggle training data.
- Record the accuracy obtained for predictions on the validation data.

I will use three different variants of this basic approach to see which works better. These are:

- Construct and train a CNN from scratch.
- Use an existing CNN model, but adapted to return only 'cat' or 'dog' classes.
- Use an existing CNN model, but fine-tune it using the Kaggle training data.

Metrics

The metric proposed to evaluate the classifier performance is simple *accuracy*. Accuracy is the number of correctly predicted class labels, divided by the total number of predictions made. This is a simple and easy-to-understand metric, and is appropriate due to the dataset being balanced and only binary classification required.

Alternative metrics sometimes used instead of accuracy include:

- *Precision*, the number of true positives out of all positives predicted.
- *Recall*, the number of true positives out of all true positives and false negatives.

- *F-1 score*, a single measure combining both precision and recall.

Precision, recall and f1 have their advantages when the dataset is imbalanced. For instance, if it consisted of 90% cat images and 10% dog, then a classifier that always predicted *cat* would achieve an apparently 90% accuracy. The recall score would be much higher than precision though, revealing the weakness in the model that accuracy alone would miss.

Another use case is when false positives and false negatives are not equivalent. An example would be a model used to detect cancer: a false positive ("told you have cancer, but you do not") is bad, but a false negative ("told you do not have cancer, but you do") is life threatening. In this case, a model with high recall (lowest numbers of false negatives) may be preferred over one with higher accuracy but poorer recall.

For this dataset, there's an equal number images of cats and dogs, and there's no difference between false positives and negatives. Precision, recall and f1-score do not offer any real advantage over accuracy in this case.

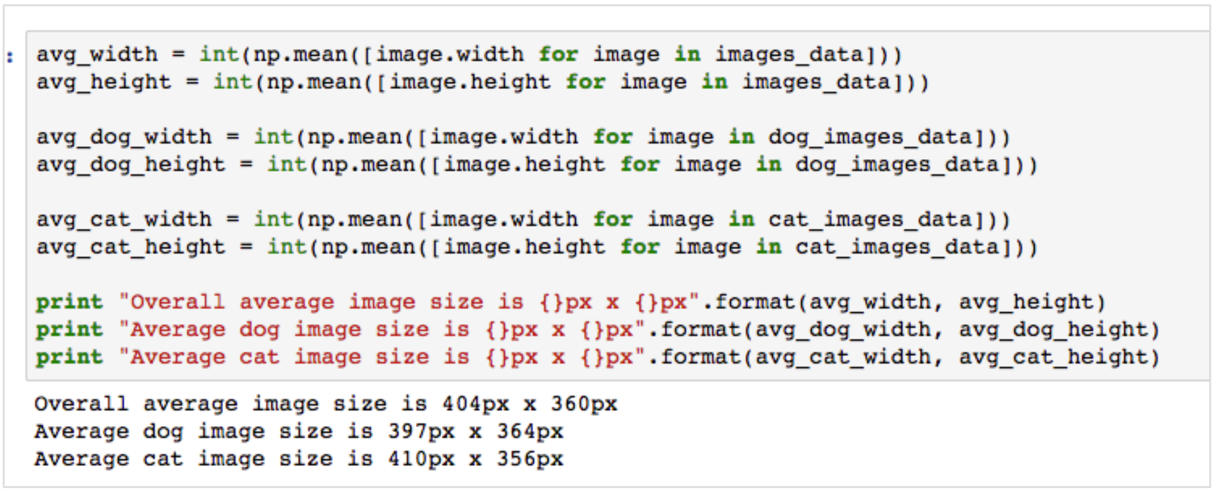
Analysis

Data Exploration

The dataset for the problem is provided by Kaggle and is available on their site (Kaggle, no date). The labelled dataset contains 25,000 images, half of cats and half of dogs. The label for each image is given by the filename, such as *dog.1.jpg*. The file sizes for the images are variable and not consistent. The majority of the images are low-resolution and poor quality, adding to the challenge in identifying them. A small percentage of the images contain more than one animal in them.

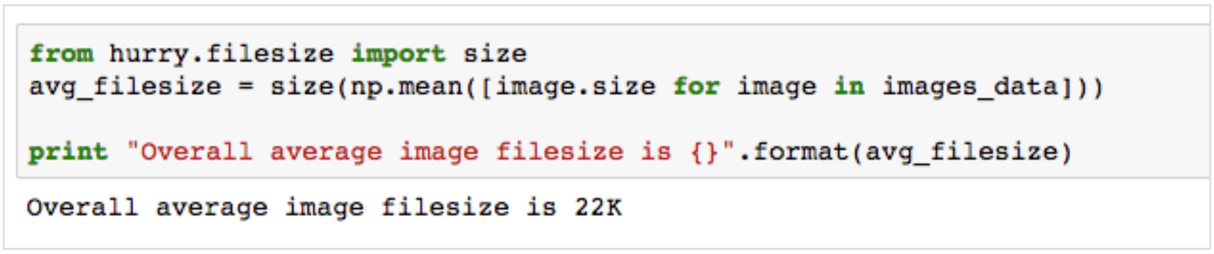
The training and test set both originally come from the Asirra dataset (Microsoft, 2007), of which the full dataset contains more than 3 million images of cats and dogs from petfinder.com. This makes it an ideal dataset for this problem, although as the images are specifically of pet dogs and cats it may be skewed towards certain breeds that are most popular as pets.

The images average a size of 404x360px, and a file size of 22KB.



```
avg_width = int(np.mean([image.width for image in images_data]))
avg_height = int(np.mean([image.height for image in images_data]))

avg_dog_width = int(np.mean([image.width for image in dog_images_data]))
avg_dog_height = int(np.mean([image.height for image in dog_images_data]))

avg_cat_width = int(np.mean([image.width for image in cat_images_data]))
avg_cat_height = int(np.mean([image.height for image in cat_images_data]))

print "Overall average image size is {}px x {}px".format(avg_width, avg_height)
print "Average dog image size is {}px x {}px".format(avg_dog_width, avg_dog_height)
print "Average cat image size is {}px x {}px".format(avg_cat_width, avg_cat_height)
```

```
Overall average image size is 404px x 360px
Average dog image size is 397px x 364px
Average cat image size is 410px x 356px
```



```
from hurry.filesize import size
avg_filesize = size(np.mean([image.size for image in images_data]))

print "Overall average image filesize is {}".format(avg_filesize)
```

```
Overall average image filesize is 22K
```

Example images of a dog and a cat from the dataset are:

Exploratory Visualization

One feature that can be noticed about the data is that the images are not of a consistent size or aspect ratio. In the above images for instance, the dog photo is in landscape orientation while the cat image is almost square.

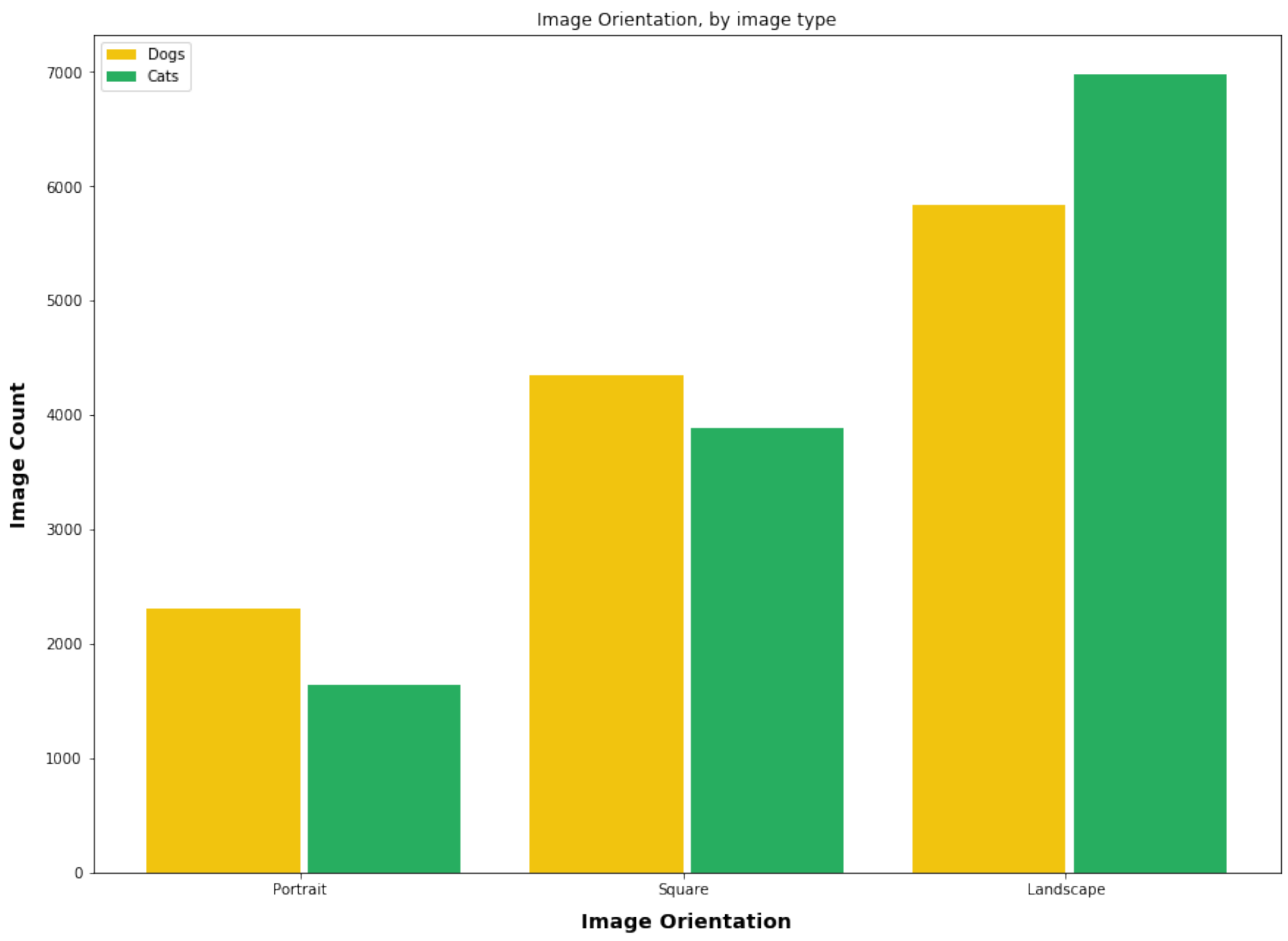
CNNs require all the training images to be of the same square size however, and so many of the images will have to be resized first. This may be a factor in how useful they are; for instance, a panorama-style image may end up being so distorted by the resizing that it is not effective as training data.

Image orientation can be visualised by calculating the aspect ratio of each image, classifying it ‘Portrait’, ‘Landscape’ or ‘Square’ accordingly, and then grouping and plotting the result.

- ‘Landscape’ is defined as images with a 5:4 or greater aspect ratio.

- ‘Portrait’ is defined as images with a 4:5 or lower aspect ratio.
- ‘Square’ is images with an aspect ratio in between these. This means that not all images in this category will have an exact 1:1 ratio, but that they are close enough that any resizing will not cause significant distortion.

Using the entirety of the training data produces a plot of the following:



There are more ‘portrait’ and ‘square’ images of dogs, and conversely many more ‘landscape’ images of cats. This suggests that cat images will have to be resized more heavily, which may affect the ability of the network to recognise them.

Algorithms and Techniques

The algorithm used for this project will be a convolutional neural network, which has out-performed all other techniques in image classification tasks for at least the last 5 years (Russakovsky et al, 2015). CNN consists of layers of neurons, which receive weighted inputs and pass them through an activation function. Unlike in conventional neural networks, where each neuron in a layer is connected to every neuron in the previous layer, in CNNs each neuron is only connected to a small region of their input volume. This small region is slid ('convolved') across all positions of the input volume to produce an activation map. These activation maps are stacked together to produce an output volume. The final layer is a fully connected layer which outputs a predicted class label and probability given the input image.

This process makes CNNs very powerful for image recognition, with the lower layers capable of representing general abstract shapes, and the upper layers specific features. CNNs work best when trained with lots of data; during the training process the weights used to adjust the importance of the inputs are constantly updated by a process of back-propagation. The deeper the network and the higher the number of neurons the better, however more sophisticated architectures take a long time to train.

A well-known CNN architecture available to use is the VGG 16 model, which comes with its weights pre-trained on millions of images. Re-using these may work better than building and training a simpler CNN from scratch, so both approaches will be tried and compared.

Some of the default parameters and variables used will be:

- *The input size of the image*: Images are resized to 224x224.

- *The number of training and validation samples*: This will be split 23,000 training images and 2000 validation images. Each will be half of cats, half of dogs.
- *Convolutional filter size and stride size*: These are set to 3x3 with a stride of 1x1, in accordance with current common practice.
- *Loss function*: “Binary cross-entropy” is being used as the loss function, as the desired end result is a binary classification.

Benchmark

A basic benchmark for this project would be whether it can give better results than could be obtained by chance. With the two output classes of either 'cat' or 'dog', random guessing would be expected to give an accuracy of 50%. Therefore, we would expect this classifier to be significantly above 50% in order to prove its effectiveness. This is also the benchmark set by Kaggle, which currently lies around position ~925 of 1200 in the leaderboard.

A more challenging benchmark would be to beat the 82.7% accuracy achieved on a similar dataset by Golle (2008). That score was achieved using an older SVM classifier, and so it should be possible to improve upon it by using convolutional neural networks and modern deep learning techniques.

Methodology

Data Preprocessing

The data will be preprocessed in a couple of different ways.

- *Image Resizing*: All images in the CNN need to be the same size. This is not the case initially (see "Exploratory Visualisation), so they will need to be resized first. They will be resized to 224x224 pixels, as this is the same setting as the VGG 16 network was originally trained with.

- *Image Normalisation*: The RGB channels of a pixel have values in the 0-255 range. To make these easier to work while using a small learning rate, they are scaled down 255 times so all values are bounded between 0 and 1.

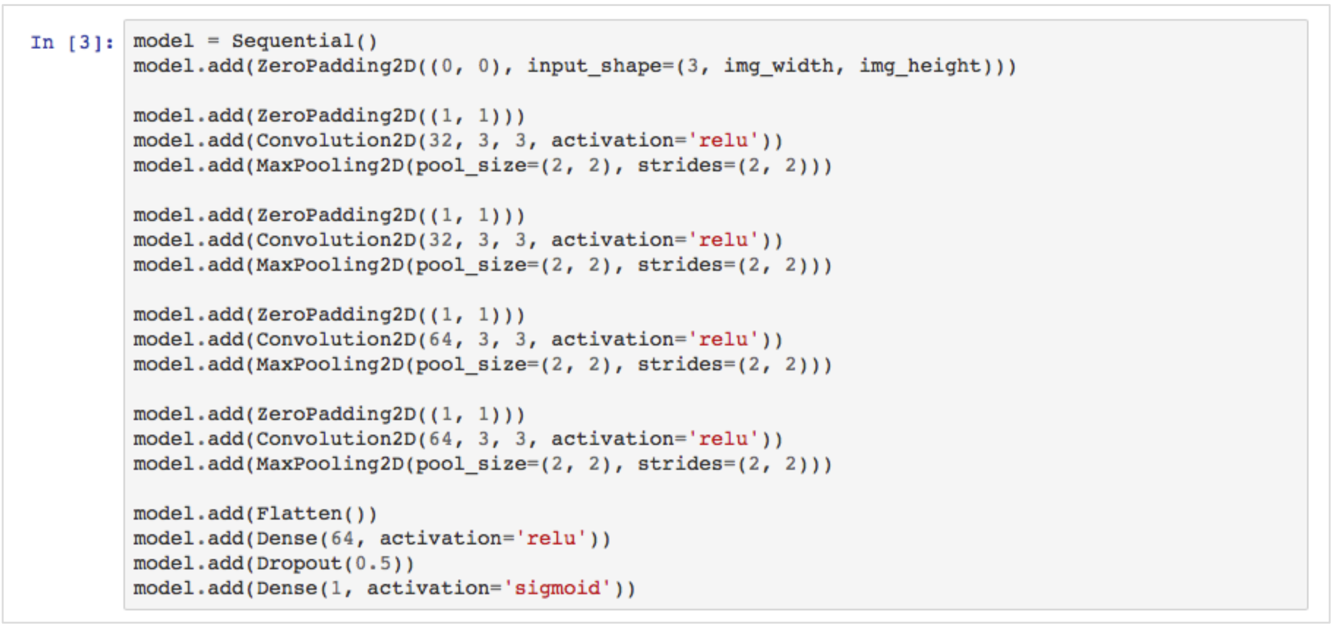
- *Directory Structure*: The Keras framework requires separate directories for each image class, so they will be structured this way accordingly.

An additional, but optional, form of image pre-processing is *data augmentation*. This helps the model to generalize better by automatically creating more, and more varied, examples training images. The augmented images are created by some form of transforming the existing images, such as by rotating, zooming, scaling, flipping and/or horizontally or vertically shifting them. Whether data augmentation is needed depends on the accuracy achieved, and how long the training takes.

Implementation

The first step in the implementation was splitting the data into separate training and validation sets, each of half cats, half dogs. Three different CNN approaches were then tried, using the Keras framework and Theano backend to create the networks. An Amazon AWS P2 instance with a Tesla K80 GPU was used to train the networks, due to the large amount of input data and training time needed. All coding was done and is documented in Jupyter notebooks, along with detailed comments.

The first approach (the Jupyter notebook is in the *implementation/simple_convnet* folder) was to construct a simple convolutional neural network.



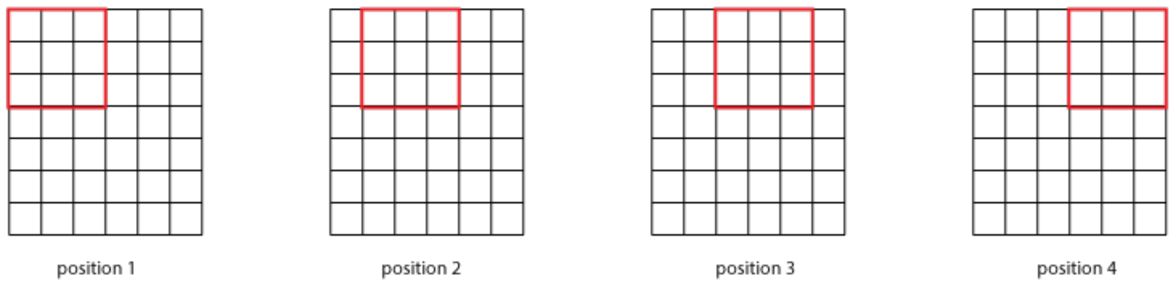
```
In [3]: model = Sequential()
        model.add(ZeroPadding2D((0, 0), input_shape=(3, img_width, img_height)))

        model.add(ZeroPadding2D((1, 1)))
        model.add(Convolution2D(32, 3, 3, activation='relu'))
        model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))

        model.add(ZeroPadding2D((1, 1)))
        model.add(Convolution2D(32, 3, 3, activation='relu'))
        model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))

        model.add(ZeroPadding2D((1, 1)))
        model.add(Convolution2D(64, 3, 3, activation='relu'))
        model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))

        model.add(ZeroPadding2D((1, 1)))
        model.add(Convolution2D(64, 3, 3, activation='relu'))
        model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))

        model.add(Flatten())
        model.add(Dense(64, activation='relu'))
        model.add(Dropout(0.5))
        model.add(Dense(1, activation='sigmoid'))
```

A simple convolutional neural network implementation in Keras.

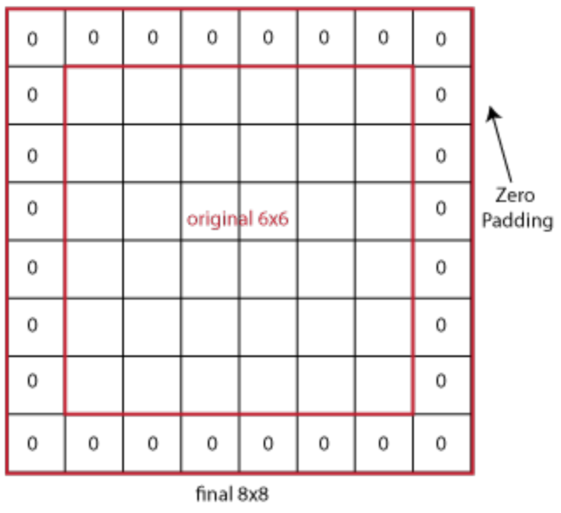
The types of layers used here are:

Convolution2D layers, which apply a 3x3 filter over all unique positions in the image to produce a new feature map. These are stacked on top of each other, the greater the number the more features they can represent.

ZeroPadding2D layers, used to counteract the input size shrinking that would otherwise occur when using a 3x3 filter. For instance, the image below shows why a 3x3 filter on an image size of 6x7 results in an output of 4x5.

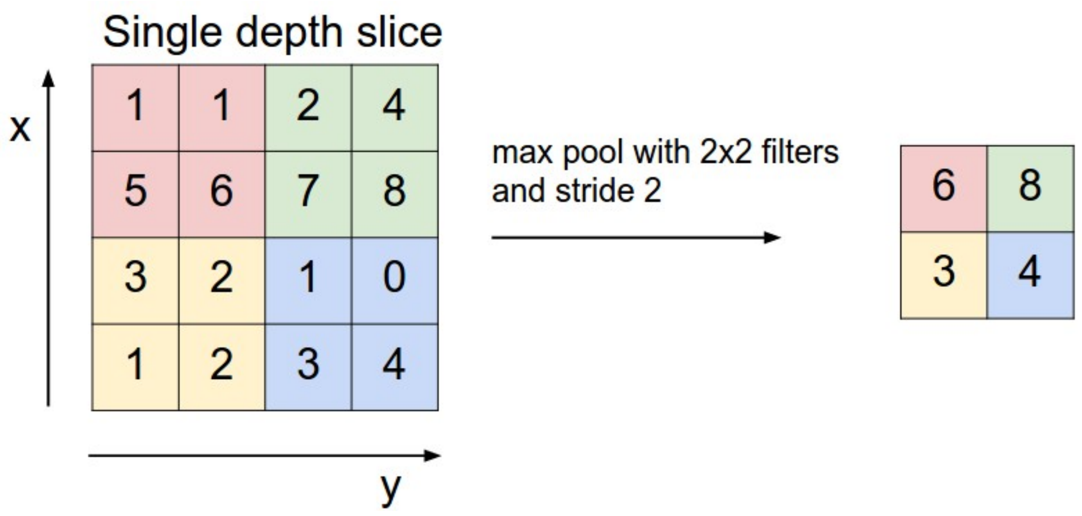


(Jain, 2016)



Zero padding adds extra blank space on all sides, resulting in the number of unique positions for filter convolutions to be the same size as the input size.

MaxPooling2D layers, which are used to down-sample the input and reduce it by half each time. This means subsequent convolutions can represent larger and larger blocks of the input. This is why lower layers of the CNN represent general abstract features and shapes, and higher level layers are much more specific. Max pooling works by taking the single highest input value in a small block (commonly 2x2), and discarding the remaining values.



Example of max pooling.

(Karpathy, 2016)

Dense layers are used to represent combinations of features together, and also to generate the final class label.

This model consists of 4 different convolutional layers, each followed by a max pooling layer used to half the input space. The first two convolutional layers are of 32 filters each, the latter two of 64 filters each. The top layers of the network consisted of two dense layers, which generated the predictions. It was then trained on all 23,000 training images, and accuracy scores obtained by testing against the validation data.

The second approach (the Jupyter notebook is in the *implementation/vgg16_new_top_layers* folder) was to download and use the VGG 16 network architecture and weights. However, as VGG 16 is designed to classify images into the 1000 ImageNet classes, it is necessary to replace the final dense layers.

After importing the weights into this model, it is run on both the training and validation sets to generate the intermediate features data. To adapt this to return the desired class predictions, the intermediate features are then used as the input to a small model that simply uses dense layers to generate the predictions.

Refinement

Automatic hyper-parameter optimization using grid search on CNNs is difficult, because of the large training time. However, trying different combinations showed larger batch sizes resulting in faster running times and also cause a small increase in accuracy. Validation accuracy generally plateaued past 10 epochs of training.

Validation accuracy, per batch size and number of epochs run

	5	**10**	**20**	**50**	**TIME**
32	93.45%	93.75%	93.60%	92.75%	**300S**
64	92.55%	92.40%	92.50%	93.70%	**150S**
128	92.00%	94.00%	94.30%	94.00%	**100S**

An additional refinement approach was by improving the model used. Instead of training a separate model for the top layers, they were joined into the existing VGG 16 model. The Jupyter notebook for this approach is in the *implementation/vgg16_fine_tune* folder.

This combined model was then re-trained (fine-tuned) on all the images. This produce an improvement over the second approach (shown below), where the convolutional layer weights remained those of the download VGG 16 weights and were not further trainable.

Results

Model Evaluation and Validation

These scores are the results after 20 epochs of training each, using 23,000 training images and 2,000 validation images.

Model Type	**Training Accuracy**	**Validation Accuracy**
Simple ConvNet	92.24%	86.31%
VGG 16 + new top layers	97.44%	92.90%
VGG 16 + fine tuning	99.70%	95.80%

All models produced impressive results, with the fine-tuned VGG model being the winner. I believe this model is both robust and generalizes well to unseen data.

Unseen data: The validation set was not used at all during training, so the validation set accuracy score is a good reflection of its performance on unseen data. Despite the high validation set accuracy, the models are likely over-fitting to some degree, as training set accuracy was consistently a few percentage points higher still. Further refinement could reduce the overfitting tendency and perhaps bring up the validation accuracy still further.

Robustness: Input images in both training and validation sets have little to no consistency in terms of resolution, photo quality, lighting, the position or angle of the subject and so on. Nevertheless, it is able to consistently classify them with high accuracy.

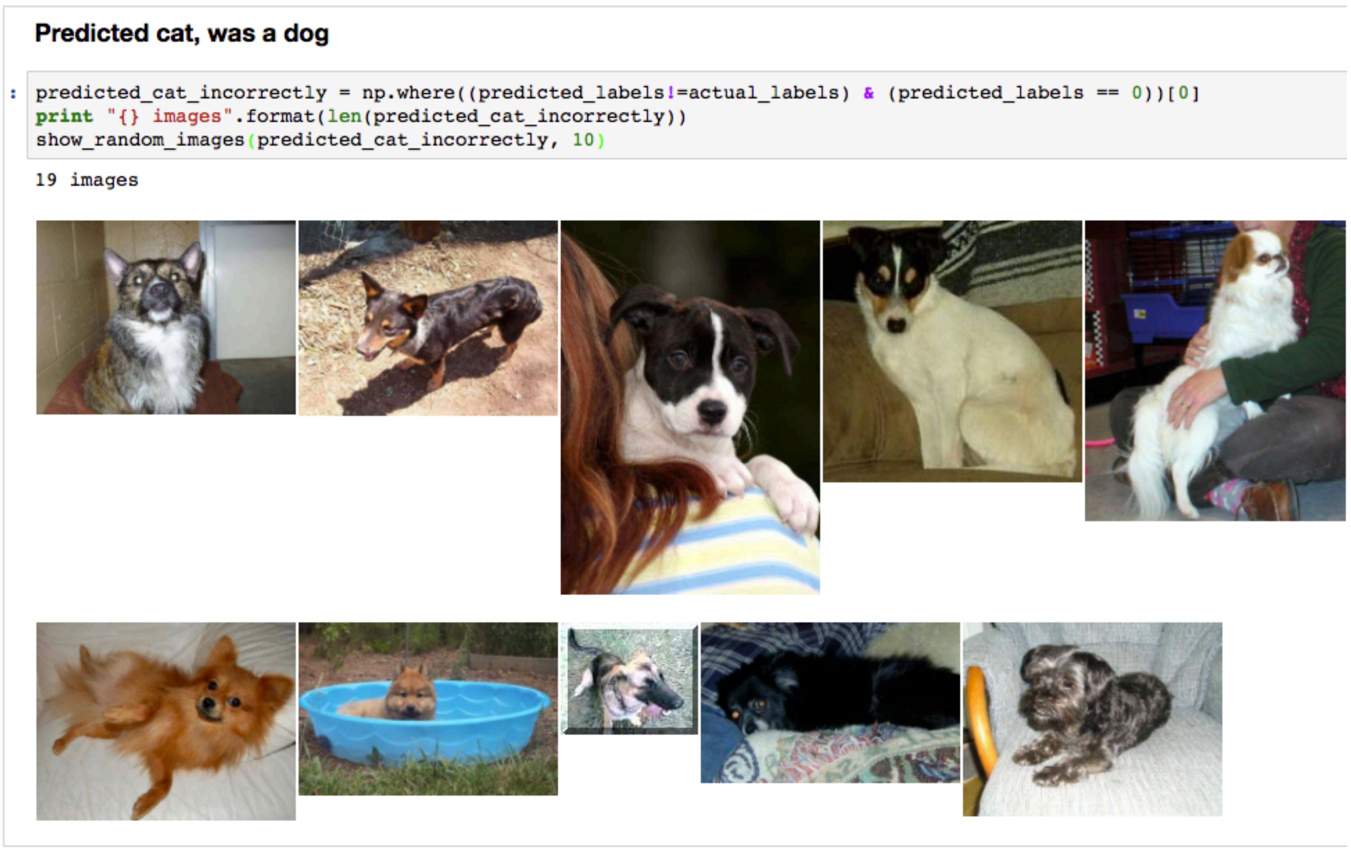
Justification

The original benchmarks for this project were to beat 50% random chance accuracy, and to beat 82.7% accuracy that was the state-of-the-art before deep learning and CNNs. All three models easily beat these benchmarks, and offer scope for further improvement still. These scores demonstrate the effectiveness of convolutional neural networks for image classification tasks, and that nowadays a CAPTCHA based on the idea that only humans can reliably distinguish dogs vs cats would be very inadvisable.

Conclusion

Free-Form Visualization

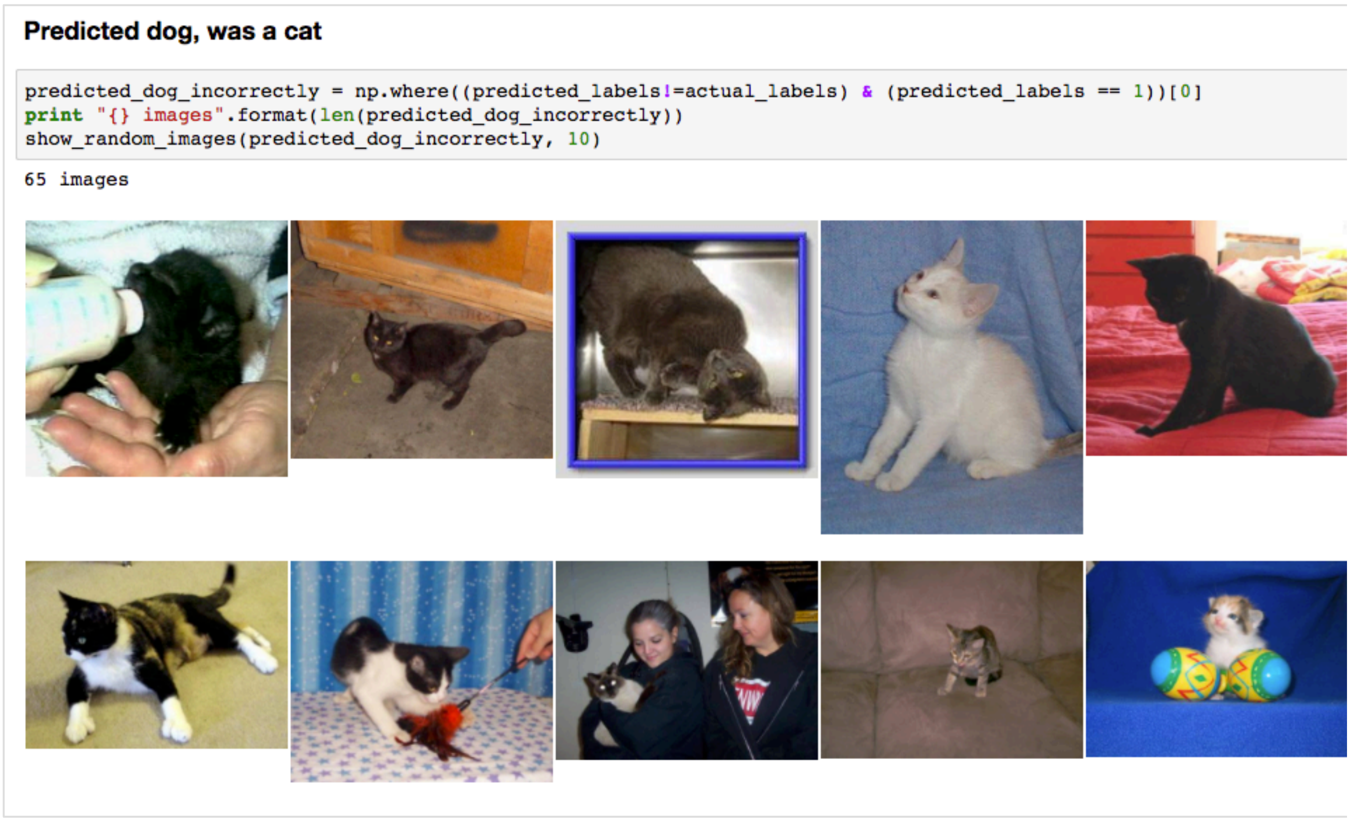
With accuracy at almost 96%, it's interesting to examine the images that were incorrectly classified to see what made them different. (Visualization code is in *implementation/vgg16_fine_tune* folder, at the bottom of the notebook.)



Predicted cat, was a dog

```
predicted_cat_incorrectly = np.where((predicted_labels!=actual_labels) & (predicted_labels == 0))[0]
print "{} images".format(len(predicted_cat_incorrectly))
show_random_images(predicted_cat_incorrectly, 10)
```

```
19 images
```



Predicted dog, was a cat

```
predicted_dog_incorrectly = np.where((predicted_labels!=actual_labels) & (predicted_labels == 1))[0]
print "{} images".format(len(predicted_dog_incorrectly))
show_random_images(predicted_dog_incorrectly, 10)
```

```
65 images
```

Many of these mistakes seem ‘understandable’, in that the they include images where the subject is small or unclear, images of dog breeds that look like cats and images of cats in poses more associated with dogs. Visualizing this can help shed light on how to improve accuracy further; for instance, if there were more small dog breeds in the training data, it could help to stop them being misidentified as cats.

Reflection

The process for this project can be summarized as:

1. Retrieve the data from Kaggle.
2. Structure it according to Keras requirements.
3. Create a model, test it locally until it seems to be working correctly.
4. Upload to Amazon p2.xlarge server, and train the model.
5. Save the weights, and refine based on results.

Overall, I am happy this model is very suitable for solving this or similar image classification problems, and it out-performed what I had originally expected would be possible. A couple of interesting aspects / difficulties I had with the project were:

- I did not initially realize the importance of image normalization (scaling down by 255), and did not implement it. Trying to train without doing this would produce only minimal improvement over 50% accuracy even after 10+ epochs of training, which was frustrating for a while.
- Unlike previous projects in the nanodegree, just training the models on my notebook computer was far too slow to be practical. It needed a machine with a powerful GPU to be able to complete them in a reasonable time. This meant becoming familiar with using Amazon EC2, and finding out how to run Jupyter notebooks remotely, how to transfer large amount of data back and forth etc. I'm pleased to have the chance to do this, as it will be a useful skill for me going forward.

Improvement

Although the chosen model gets high accuracy scores, it's noticeable that the training set accuracy is a few percent higher than the validation set accuracy. This suggest the model is likely over-fitting to the training data, and that the validation accuracy could be improved still further if the model could be made to generalize better. Potential ways of doing this would be:

- Using *data augmentation* to automatically generate versions of the images that are flipped, scaled, warped, zoomed and so on. This provides additional images that the training set may not originally include, and makes the model more translation-invariant.

- Increasing *dropout*, where random neurons of the network have their values set to 0. This helps to prevent overfitting as it stops the model relying too heavily on any specific feature, as it can no longer always rely on it being there. The current model uses a dropout of 50% which is already high, but perhaps going higher still would get better results.
- Adding *regularization*, which makes the network more resilient to random noise and improves its generalization.

References

Chollet, F., 2016. “Building powerful image classification models using very little data” available at https://blog.keras.io/building-powerful-image-classification-models-using-very-little-data.html

Golle, P., 2008, October. Machine learning attacks against the Asirra CAPTCHA. In *Proceedings of the 15th ACM conference on Computer and communications security* (pp. 535-542). ACM.

Kaggle, no date. "Dogs vs. Cats Redux: Kernels Edition" available at https://www.kaggle.com/c/dogs-vs-cats-redux-kernels-edition

Karpathy A., 2016. “Convolutional Networks” available at http://cs231n.github.io/convolutional-networks/

Keras, no date. "Keras Applications" available at https://keras.io/applications/
Krizhevsky, A., Sutskever, I. and Hinton, G.E., 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems* (pp. 1097- 1105).

Krizhevsky, A., Sutskever, I. and Hinton, G.E., 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in neural information processing systems* (pp. 1097-1105).

Jain A, 2016. " Deep Learning for Computer Vision – Introduction to Convolution Neural Networks" available at https://www.analyticsvidhya.com/blog/2016/04/deep-learning-computer-vision-introduction-convolution-neural-networks/

Microsoft, 2007. "Asirra: A CAPTCHA that Exploits Interest-Aligned Manual Image Categorization" available at https://www.microsoft.com/en-us/research/publication/asirra-a-captcha- that-exploits-interest-aligned-manual-image-categorization/

Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg and Li Fei-Fei. *ImageNet Large Scale Visual Recognition Challenge*. IJCV, 2015.

Simonyan, K. And Zisserman, A., 2014. Very Deep Convolutional Networks for Large-scale Image Recognition. *Arxiv Preprint Arxiv*:1409.1556.