Machine Learning Engineer Nanodegree

Capstone Project Report

Resmi Arjunanpillai

November 15, 2017

I. Definition

Project Overview

Domain Background

Computer vision as a space has seen rapid advances in just the last few years. Deep learning has created very high accuracy in recognizing images, leading to applications in multiple areas like health, self driving cars, robotics etc.

A few of the key papers in the space:

- Alexnet is the paper that jump started the current interest in CNNs. This paper by Geoffrey Hinton and others, describes a deep convolutional neural network that won the 2012 Imagenet competition.
  https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf
- VGGnet was another influential paper that demonstrated the use of smaller 3x3 filters and promoted deeper networks.
  https://arxiv.org/pdf/1409.1556v6.pdf

Considerable research has also gone into optimizing hyper parameters. A few examples

- Momentum: https://arxiv.org/pdf/1412.6980.pdf and .
- Adam optimizer: https://arxiv.org/pdf/1412.6980.pdf

Problem Statement

The Dogs vs. Cats Redux: Kernels Edition on Kaggle is a competition to differentiate images of cats from images of dogs.

https://www.kaggle.com/c/dogs-vs-cats-redux-kernels-edition

Though the problem is a well researched one, trying out many different approaches to achieve a very high level of accuracy to solve the problem can be challenging. Given the existence of several networks trained on Imagenet within Keras (VGG, Resnet etc), I would like to use transfer learning using pre-trained networks. Optimizing the learning parameters, using batch normalization, data augmentation etc would be important.

My goal on the project is to achieve a logloss of 0.1 or below on the above project.

Metrics

Evaluation Metrics

The contest uses log-loss as the evaluation metric

$$\text{LogLoss} = -\frac{1}{n}\sum_{i=1}^{n}[y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)],$$

where

- n is the number of images in the test set
- ŷ iy^i is the predicted probability of the image being a dog
- yiyi is 1 if the image is a dog, 0 if cat
- log()log() is the natural (base e) logarithm

A smaller log loss is better.

LogLoss is related to cross-entropy and measures the performance of a classification model where the prediction is a probability value between 0 and 1. LogLoss takes into account the uncertainty of the prediction based on how much it varies from the actual model.

In addition to log loss as the evaluation metric, I will also track accuracy - this measures how many dogs and cats have been categorized into the right category.

II. Analysis

Data Exploration

Data: https://www.kaggle.com/c/dogs-vs-cats-redux-kernels-edition/data

The train folder contains 25,000 images of dogs and cats. Each image in this folder has the label as part of the filename. The test folder contains 12,500 images, named according to a numeric id. For each image in the test set, the goal is to predict a probability that the image is a dog (1 = dog, 0 = cat).

I will split the training set into train/validation groups to make it easier to iterate on my model and optimize it.

There are 12,500 cat images and 12,500 dog images. The naming follows the convention cat.number.jpg and dog.number.jpg. For the validation set, I will use approx 1,000 cat and dog images each. The images are full color images with 3 channels - RGB.

The classes are balanced well and I'll maintain a similar class balance in the validation set.

Image dimensions range from 300-500 x 300-500. There are some outliers as well, but the majority of the images are within this range. Pre-trained networks available in Keras work well with images in this size range since the pre-trained networks were trained with images with a similar size from ImageNet.

Exploratory Visualization

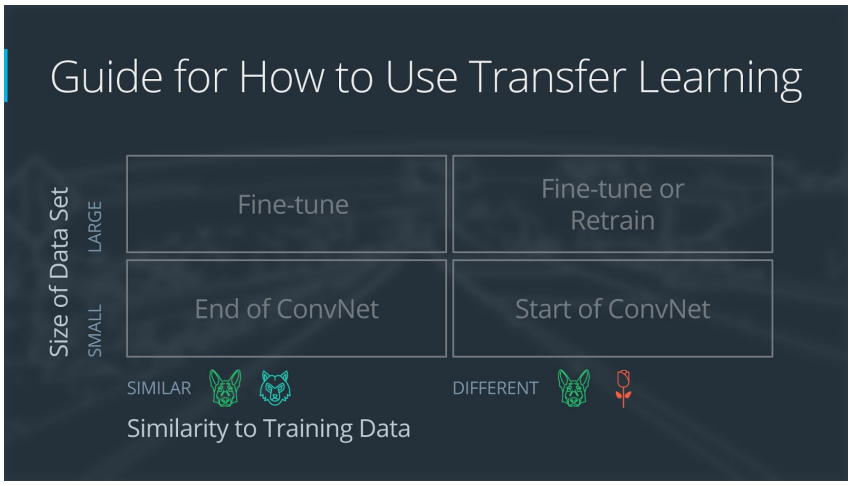
The code has a section to visualize the images themselves. Eyeballing these were helpful to see the quality of the images - 1) whether the dogs and cats took up a large portion of the images, were the images clear etc and 2) to visualize the dimensions to make sure that they were similar in size to the images from Imagenet that were used for training these networks.

This visualization was chosen since the most relevant qualities for the task at hand were the quality of the pictures and dimensions - since we I am using pre-trained networks.

Algorithms and Techniques

I used 2 pre-trained networks -- Vgg16 and Resnet50 -- in Keras.

The cats and dogs images are similar to the ImageNet images that these networks have been trained on. The cats/dogs dataset is large, but not huge.



Given these, I'll start by replacing the final fully connected layer with a layer with 2 classes. I'll train the final layer while keeping the ImageNet weights for the rest of the network.

The images will need to be pre-processed for each pretrained model. Pre-processing needed is the same for Vgg16 and Resnet50 -- remove vgg mean or resnet mean and convert RGB to BGR. Images need to be resized to 224 x 224.

With Vgg16, I will do a few more modifications to try and improve the accuracy/reduce log loss.

1. Re-train additional dense layers - not just the last layer. We have a large enough dataset to do that without considerable overfitting.
2. Change levels of dropout
    a. Dropout in neural networks throws away a percentage of activations in the dense layers thus forcing the neural network to generalize
3. Try different methods of data augmentation.
    a. Data augmentation rotates, flips and in other ways randomly modifies the input images that go through the model. This helps the neural network generalize and prevent it from overfitting to the noise in the input data.

2 and 3 above should help reduce any overfitting issues.

Picking the right learning rate[1], dropout levels and data augmentation will be key to try and improve the accuracy.

Benchmark

I'll train a VGG16 model on the data and use it as the benchmark model.

III. Methodology

Data Preprocessing

Pre-processing needed is the same for Vgg16 and Resnet50 -- remove vgg mean or resnet mean and convert RGB to BGR. Images need to be resized to 224 x 224.

Keras provides a standard way of doing this.
**
from keras.applications.vgg16 import preprocess_input

datagen = image.ImageDataGenerator(preprocessing_function=preprocess_input)
valid_datagen = datagen.flow_from_directory(valid_path, target_size = (224, 224),
class_mode = 'categorical', batch_size = batch_size)
**
Preprocess_input in the code snippet above removes vgg mean and converts RGB to BGR. target_size = (224, 224) resizes the images to 224 x 224.

The same process works for Resnet50.

In addition, for using the Keras models, the images should be organized such that all the cats are in one folder and all the dogs are in another folder. I also moved a few images to validation folders for both cats and dogs. In addition, I copied a small number of images into a sample folder, so I could quickly check that the code was working as intended[2] -- without having to train the model on the entire dataset. The code for these steps is included in a separate Jupyter Notebook - CatsVsDogsFolders.ipynb

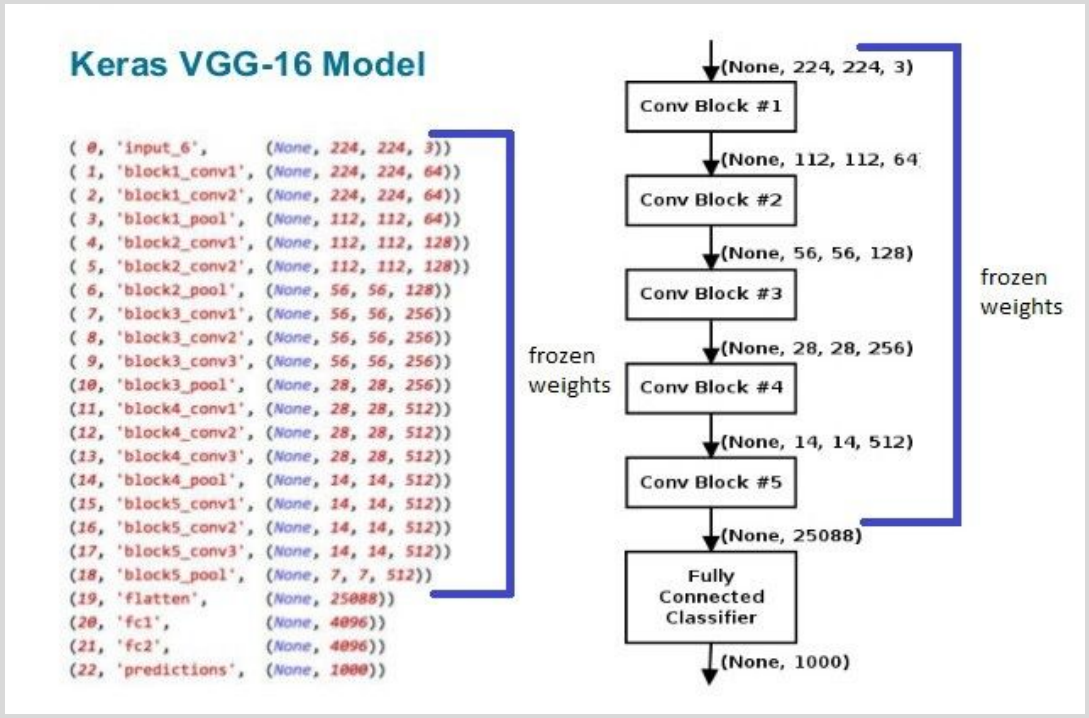
[1] http://cs231n.github.io/transfer-learning/

[2] Approach from www.fast.ai

Implementation

VGG16[3] is a convolutional neural network architecture named after the Visual Geometry Group from Oxford, who developed it. It was used to win the ILSVR (ImageNet) competition in 2014. VGG adopts a simple neural network structure. Only 3x3 convolution and 2x2 pooling are used throughout the whole network, along with 2 fully connected layers.

Keras Model of VGG16 is implemented here:
https://gist.github.com/baraldilorenzo/07d7802847aaad0a35d3



VGG16 architecture - this shows the setup when all the dense layers are re-trained

I took a few approaches to the problem.

- Fine-tune the last layer of VGG16.

[3] Very Deep Convolutional Networks for Large-Scale Image Recognition
K. Simonyan, A. Zisserman arXiv:1409.1556

In this step, built-in VGG16 was imported - this imports the model as well as the weights used for the original VGG16. Using the functional API, the last layer with 1000 nodes is removed (the 1000 nodes were used for predicting the original 1000 categories in Imagenet).

A new layer is added with 2 nodes -- to classify cats vs. dogs.

All layers, except the last layer, are frozen and the network in compiled and fitted with the cats vs dogs data.

- Re-train all the dense layers of VGG16.

2 additional layers of VGG16 are set to trainable. These are the fully connected/dense layers in the image above - fc1 and fc2.

Cats and Dogs are part of the original Imagenet categories that were used to train VGG16. So, the weights of these layers are probably close to the correct weights for predicting cats and dogs. So, a small learning rate (1e-6) is used to compile and fit the model.

One of the main difficulties of training was finding the right learning rate. The model did not train well, till the learning rate was reduced to this level.

Since we replaced the last layer, that layer must be first fine-tuned with fc1 and fc2 frozen. Starting the last layer with random weights may cause the weights of fc1 and fc2 to move too far away from the Imagenet weights. Hence the step of fine-tuning the last layer must be completed before setting fc1 and fc2 to be trainable.

- Use data augmentation

Data augmentation is used to prevent models from overfitting - when neural networks are trained over multiple epochs, they become very good at predicting the training images, but they may not generalize well.

To reduce overfitting, we use data augmentation - the data is augmented by random transformations that modify the images slightly -- so that the network does not see the same image twice.

The image.ImageDataGenerator class in Keras lets us do data augmentation.

It lets you do 1) random transformations on your image and 2) Generate augmented image batches via .flow or .flow_from_directory.

Here is an example of the data augmentation code used in this program

datagen_aug2 =
image.ImageDataGenerator(preprocessing_function=preprocess_input, rotation_range=40, width_shift_range=0.2, height_shift_range=0.2, shear_range=0.2, zoom_range=0.2, horizontal_flip=True, fill_mode='nearest')

Here are a few of the augmentation options that were used[4]

- rotation_range is a value in degrees (0-180), a range within which to randomly rotate pictures
- width_shift and height_shift are ranges (as a fraction of total width or height) within which to randomly translate pictures vertically or horizontally
- shear_range is for randomly applying shearing transformations
- zoom_range is for randomly zooming inside pictures
- horizontal_flip is for randomly flipping half of the images horizontally
- fill_mode is the strategy used for filling in newly created pixels, which can appear after a rotation or a width/height shift.

For data augmentation, I used 2 types of data augmentation. In the first round, more limited augmentation was used and in the second round, greater augmentation was used. In practice, typically some types of augmentation will work better than others and trial and error is often required.

- Adding dropout.

Another method of reducing overfitting is to add dropout. Dropout randomly deletes a percentage of activations in the dense layers. This forces the network to generalize its training and prevents the network from overfitting

The original vgg16 model included dropout, but the keras version does not.

[4] www.keras.io

Hence I imported the built-in VGG16 without the dense layers from Keras and added 2 dense layers with dropout in between.

Two levels of dropout were tried -- 0.5 and 0.7.

Here is the code snippet for creating a model with varying levels of dropout.

```
def m_build (dropout, gen, filepath):
    last2 = base_model.layers[-3].output
    x = Dropout(d)(last2)
    x = Dense(4096, activation='relu')(x)
    x = Dropout(d)(x)
    predictions = Dense(2, activation='softmax')(x)
    drop_model = Model(inputs=base_model.input, outputs = predictions)

    for layer in drop_model.layers[20:]:
    layer.trainable = True
    for layer in drop_model.layers[:20]:
    layer.trainable = False

    m_compile(drop_model, lr = 0.00001)
    m_fit(drop_model, gen, filepath)

    return drop_model

filepath = results_path + "sweights_drop1.{epoch:02d}-{val_loss:.2f}.hdf5"
drop_model1 = m_build(0.7, gen, filepath)
```

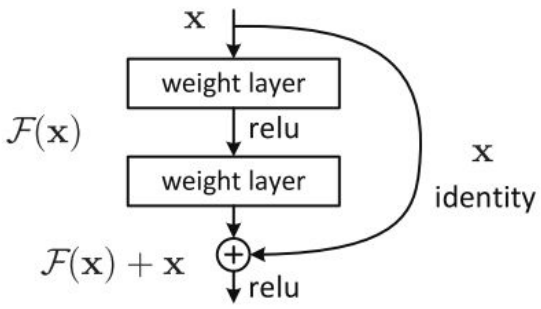
Training Resnet50 on the Cats vs Dogs Data

I also fine-tuned the last layer of a Resnet50[5] model on the cats and dogs data. The process for fine-tuning the last layer of the Resnet50 follows the same steps as fine-tuning the last layer of VGG16.

[5] https://arxiv.org/abs/1512.03385
Deep Residual Learning for Image Recognition

ResNet stands for Residual Network - a network that does residual learning. In residual learning, instead of trying to learn features, the model tries to learn a residual. Residual is a subtraction of the feature learned from input of that layer.

As convolutional networks become deeper, training of the network becomes more difficult and accuracy can start plateauing or degrading. ResNet introduces a so-called “identity shortcut connection” that skips one or more layers, as shown in the following figure:



The hypothesis is that letting the stacked layers fit a residual mapping is easier than letting them directly fit the desired underlying mapping. ResNet50 is a 50 layer Residual Network.

Here are the results from the first few approaches

Model	Training accuracy	Training Loss	Validation Accuracy	Validation loss	Num_ epochs to attain final model
VGG16 with finetuned last layer	0.9777	0.0672	0.9840	0.0474	1
VGG16 with multiple layers tuned	0.9906	0.0294	0.9830	0.0502	1
ResNet50	0.9855	0.0393	0.9795	0.0518	5

Refinement

The refinements tried are discussed already in the previous section. More details included here.

VGG16 with the fine-tuned last layer and VGG16 with all dense layer re-trained, quickly overfit the data (training error was lower than validation error). Hence it was important to try methods that reduced overfitting -- data augmentation and adding dropout.

If the original attempts were not overfitting, it would not have made sense to try the above approaches.

Additional refinements included tuning of some of the hyper parameters. Learning rate had to adjusted to make some of the models train. For e.g.:, while the last layer was tuned with a learning rate of 1e-2, fine-tuning additional dense layers required a much lower learning rate 1e-6.

I also tried different levels of data augmentation and dropout levels to see what worked best.

IV. Results

Model Evaluation and Validation

Results of different approaches are summarized below.

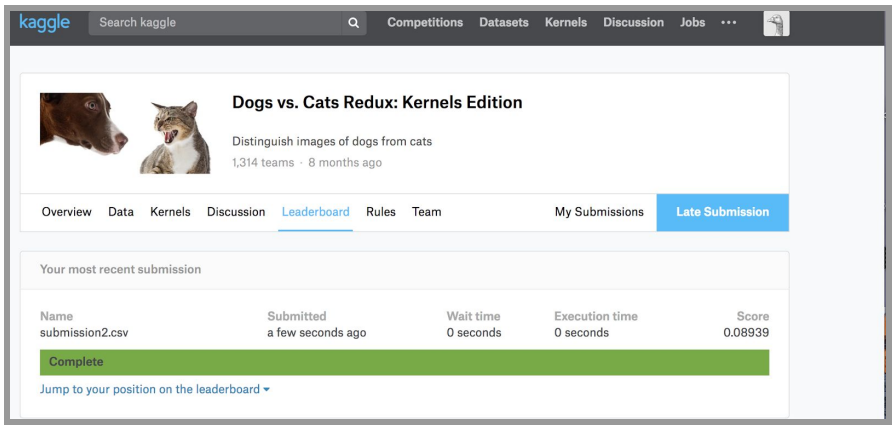
Model	Training accuracy	Training Loss	Validation Accuracy	Val_loss on best model	Num_ epochs to final model
VGG16 with finetuned last layer	0.9777	0.0672	0.9840	0.0474	1
VGG16 with multiple layers tuned	0.9906	0.0294	0.9830	0.0502	1
ResNet50	0.9855	0.0393	0.9795	0.0518	5
VGG with some data aug	0.9900	0.0264	0.9855	0.0439	4

VGG with more data aug	0.9872	0.341	0.9860	0.393	12
VGG dropout 0.5	0.9890	0.0754	0.9865	0.1093	3
VGG dropout 0.7	0.9210	0.7308	0.9820	0.1594	1

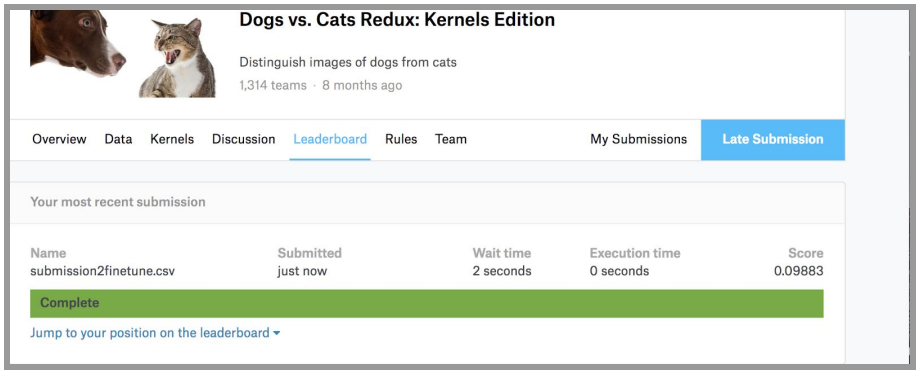
The VGG model with a higher level of data augmentation worked the best. Second best was VGG with just the last layer fine-tuned.

Final predictions on both these models were generated with the test data and the predictions were uploaded to Kaggle. The models performed well on the test data.

Logloss on predictions made by VGG with data augmentation was 0.08939



Logloss on the VGG model with last layer fine-tuned was 0.09883



My goal on the project was to achieve a logloss of 0.1 or lower on the test set and both these models achieve that.

Checking on both validation and training sets shows that the model generalises well to unseen data.

The test set is large enough to show the robustness of the model. The model works well to classify the large number of images used in the test set and achieved a good log loss value.

Justification

The goal of the model was to achieve a log loss of 0.1 or lower. My benchmark model was VGG16.

2 models achieved a log loss lower than the 0.1 goal. And the model with dropout beat the logloss result of the benchmark model on the validation set and the Kaggle test set.

V. Conclusion

Free-Form Visualization

Here, I visualized the images to which augmentation was applied. Augmentation is an important criteria for my model as well as for neural networks in general - enabling longer training of networks without overfitting. Please see augmented images in the Jupyter Notebook.

Visualizing the augmented images enabled me to ensure that the augmentation was generating images that are reasonably close to cats and dogs images that appear in real-life.

Reflection

In this project, I used multiple approaches to solve the cats vs dogs Kaggle competition.

The baseline approach tried using pre-trained VGG16 to solve the problem and additional approaches included training additional layers, adding dropout and augmentation. I also tried a newer neural network Resnet50 to solve the same problem.

Cats and dogs are very similar to the types of images used to train Imagenet. Hence, several of the attempts to unfreeze additional layers did not result in better models - the weights on these layers are already close to optimal for the cats vs dogs dataset and training them did not improve them.

This was true for the models with the dropout as well - the new dense layers added in the models with dropout had to be trained from scratch which prevented the model from outperforming the benchmark.

This model can be used for any similar image classification task.

Improvement

Many potential improvements could be made to the model that can improve the logloss and accuracy.

A few quick approaches:

- More data augmentation attempts with trial and error.
- Train the data augmented models for longer and possibly lower the learning rate.
- Use data augmentation to train the VGG model with last layer fine-tuned
- Pre-calculate the bottleneck activations for the model (activations after the final convolution layer) which could lead to faster training and testing of more models. This however will not work for models where we want to try data augmentation.
- Try additional, newer pre-trained models like InceptionNet

In general, more experimentation with learning rates, dropout rates and data augmentation could produce better results. Main challenge in implementing these is the computational time and cost involved.