Avishek Kedia

Capstone Project
Machine Learning Engineer
Nanodegree

November 22, 2016

Distinguish images of dogs from cats

Definition

Project Overview

Image recognition have received a large amount of attention over the last few decades. We can see it through hundreds of books published on the subject and the growing of new big train in Machine Learning: Deep Learning. By using a hierarchy of numerous artificial neurons, deep learning can automatically classify images with a high degree of accuracy. Thus, neural networks can recognize different species of cats, or models of cars or airplanes from images.

In this project, we focus on classify whether images contain either a dog or a cat. We have built a cat/dog classifier using a deep learning algorithm called Convolutional Neural Network(CNN). The application uses a classifier trained using the Kaggle dataset. This project was inspired by Kaggle competition.

Problem statement

The goal is to classify whether a given image contain either a dog or a cat, the tasks involved are the following:

1. Download Dogs and Cats data.

First, we need to download the 2 datasets (train.zip and test.zip) from the competition page:

- train.zip contains labeled cats and dog's images that we will use to train the model.
- test.zip contains unlabeled cats and dog's images that we will use to classify to either dog or cat using the trained model.

2. Train the CNN on the training data
3. Make prediction on the testing data

Metrics

The accuracy metric for binary classifier with formula is used:

$$accuracy = \frac{true\ positives + true\ negatives}{dataset\ size}$$

- True positives in this case is when the input is dog image and the CNN predict 1= dog as output; or the input animal is cat and our algorithm predict 0=cat.
- True negatives in this case is when the input is dog image and the CNN predict 0= cat as output; or the input animal is cat and our algorithm predict 1=dog.

This metric is used because, each dataset images are labeled with the label dog or cat(classes). Given our dataset, a classification (the output of a classifier on that set) gives two numbers: the number of positives and the number of negatives, which add up to the total size of the set. To evaluate the classifier, one compares its output to original reference. So we got Classification accuracy which is the number of correct predictions made as a ratio of all predictions made.

Analysis

Data Exploration

The dataset (from Kaggle dataset) contains 25,000 labeled images of dogs and cats; and the te sting data contains 12,500 not labeled images. The testing data are not labeled because these i mages are used to predict if the corresponding one is cat or dog. The images are colored and h ave different pixels' size. From kaggle we can see that Mean aspect ratio is: 1.15885238297 .

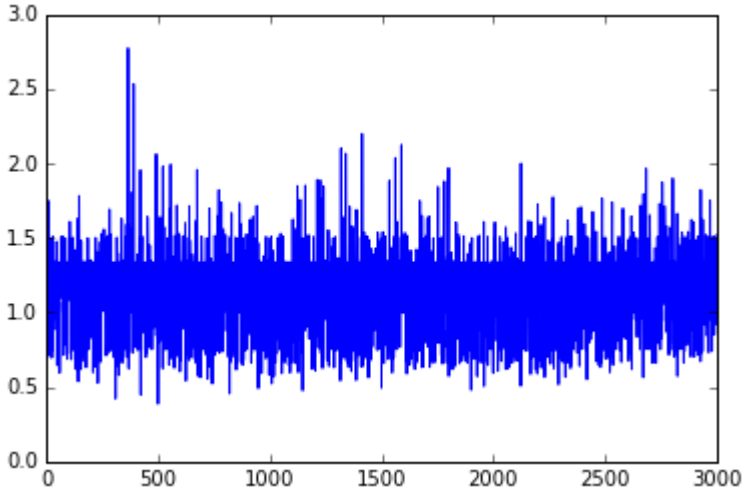

Figure 1: Aspect ratio (width/height), mean width and mean height

The following images of cat and dog with different size. From kaggle we have seen that there are only t wo images in the training set that are bigger than 500x500.

Figure 2: Images from the Kaggle dataset

Because of this difference, we have normalized our dataset as follow:

- Run histogram equalization on all training images. Histogram equalization is a technique for adjusting the contrast of images.
- Resize all training images to a 227x227 format.
- Divide the training data into 2 sets: One for training (5/6 of images) and the other for validation (1/6 of images). The training set is used to train the model, and the validation set is used to calculate the accuracy of the model.
- Store the training and validation in 2 LMDB databases. train_lmdb for training the model and validation_lmbd for model evaluation.

Exploratory visualization

The training set contains equals numbers of cats and dogs images i.e. 12,500 for each category. The following figure shows that the Mean width is 406.815333333 and the Mean height: 361.122333333. Some images are horizontally stretched, others are vertically stretched.

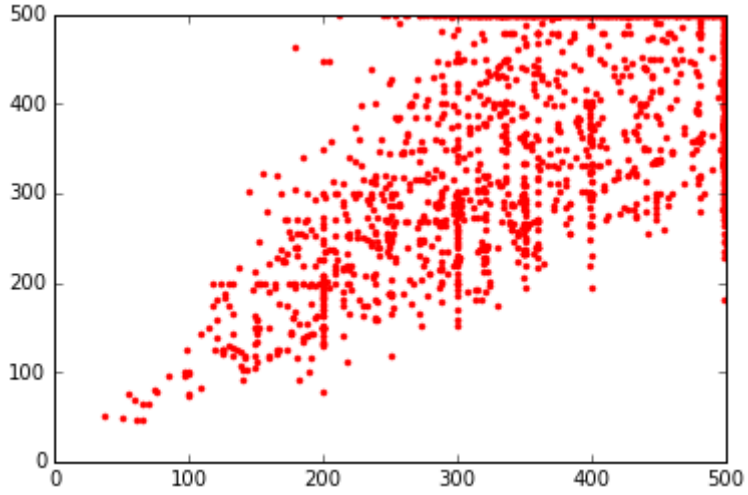

Figure 3: Mean width/height from the Kaggle dataset

In addition, we note that the mean aspect ratio varies so much. These two conditions involved us to resize all the images as described in the previous section. We want to resize the images to the same width and height for better training result and to avoid overfitting. The next reason to resize the images is because the shape of images influences the CNN algorithm. We have chosen the size format 227x227; It depends on the size of our network and our GPU. More the size format of images is large, more the time to process it is requiring.

Algorithms and techniques

The classifier is a Convolutional Neural Network, which is an algorithm for image processing tasks, including classification. CNN needs a large amount of training data for better learning; the cat's dog's datasets are big enough. The algorithm outputs an assigned probability for each class; this can be used to reduce the number of false positives using a threshold. When a new (unseen) image is input into the CNN, the network would go through the forward propagation step and output a probability for each class (for a new image, the output probabilities are calculated using the weights which have been optimized to correctly classify all the previous training examples).

The following parameters can be tuned to optimize the classifier:

- Classification threshold (see above)
- Training parameters
  - Maximum Iteration (number of epochs)
  - Solver mode (GPU or CPU)
  - Batch size (how many images to look at once during a single training step)
  - Solver type (what algorithm to use for learning)
  - Learning rate policy (learning rate policy)
  - Learning rate (how fast to learn; this can be dynamic)
  - Weight decay (prevents the model being dominated by a few "neurons")

- Momentum (takes the previous learning step into account when calculating the next one)

❖ Neural network architecture
- Number of layers
- Layer types (convolutional, fully-connected, or pooling)
  - **Convolution layer**

This layer consists of a set of learnable filters(kernels) that we slide over the image spatially, computing dot products between the entries of the filter and the input image. The filters extend to the full depth of the input image. For example, if we want to apply a filter of size 5x5 to a colored image of size 32x32, then the filter should have depth 3 (5x5x3) to cover all 3 color channels (Red, Green, Blue) of the image. These filters will activate when they see same specific structure in the images.

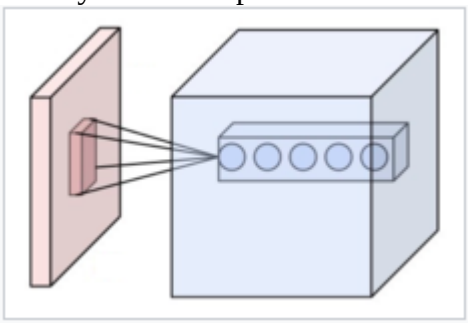

Figure 4: Neurons of a convolutional layer, connected to their receptive field

(source: Wikipedia)

  - **Pooling layer**

Pooling is a form of non-linear functions. The goal of the pooling layer is to progressively reduce the spatial size of the representation to reduce the number of parameters and computation in the network, and hence to also control overfitting. There are several functions to implement pooling among which max pooling is the most common one. Pooling is often applied with filters of size 2x2 applied with a stride of 2 at every depth slice. A pooling layer of size 2x2 with stride of 2 shrinks the input image to a 1/4 of its original size.

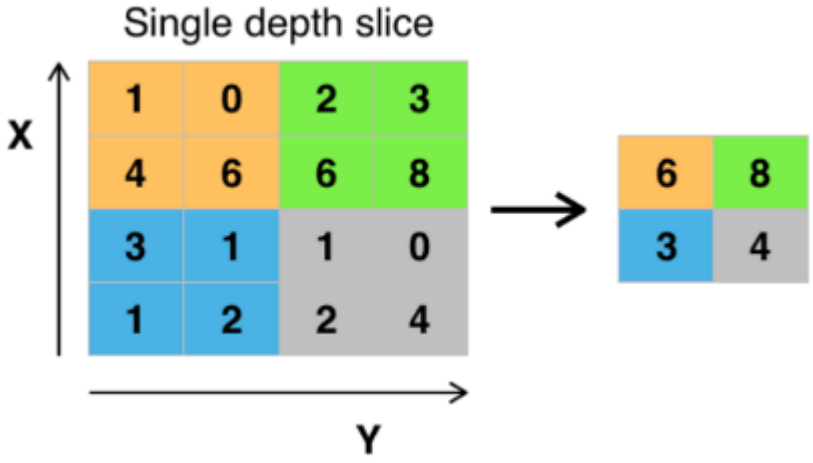

Figure 5: Max pooling with a 2x2 filter and stride = 2

(source: Wikipedia)

- **Fully connected**

  Finally, after several convolutional and max pooling layers, the high-level reasoning in the neural network is done via fully connected layers. Neurons in a fully connected layer have full connections to all activations in the previous layer. Their activations can hence be computed with a matrix multiplication followed by a bias offset.

- Layer parameters

The **figure 8** represent an example of CNN architecture. We go into more details below, but a simple ConvNet for CIFAR-10 classification could have the architecture [INPUT - CONV - RELU - POOL - FC]. In more detail:

- INPUT [32x32x3] will hold the raw pixel values of the image, in this case an image of width 32, height 32, and with three color channels R,G,B.
- CONV layer (*i.e convolution layer*) will compute the output of neurons that are connected to local regions in the input, each computing a dot product between their weights and a small region they are connected to in the input volume. This may result in volume such as [32x32x12] if we decided to use 12 filters.
- RELU layer will apply an elementwise activation function, such as the max(0,x)max(0,x) thresholding at zero. This leaves the size of the volume unchanged ([32x32x12]).
- POOL layer will perform a down sampling operation along the spatial dimensions (width, height), resulting in volume such as [16x16x12].
- FC (*i.e. fully-connected*) layer will compute the class scores, resulting in volume of size [1x1x10], where each of the 10 numbers correspond to a class score, such as among the 10 categories of CIFAR-10. As with ordinary Neural Networks and as the name implies, each neuron in this layer will be connected to all the numbers in the previous volume.

In this way, CNN transform the original image layer by layer from the original pixel values to the final class scores.

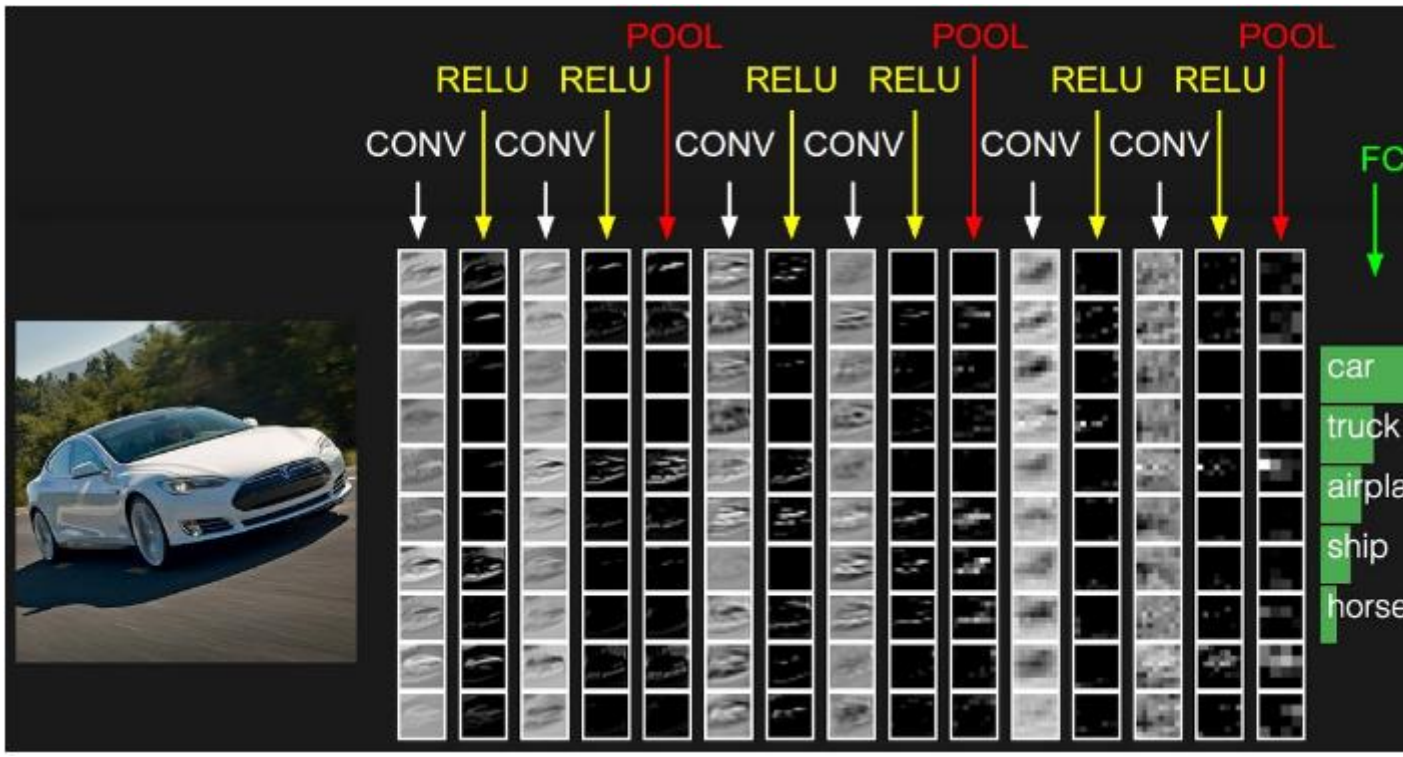

Figure 8: example Convolutional Neural Network architecture (source: cs231n)

- Preprocessing parameters (see the Data Preprocessing section)
  - Image size (image width, image size)
  - Image, label

Benchmark

To create an initial benchmark for the classifier, I used the bvlc_reference_caffenet model which is a replication of AlexNet with a few modifications.The "standard" AlexNet architecture (Figure. 9) achieved the best accuracy, around 0.97223.

Methodology

Data Preprocessing

The preprocessing done consists of the following steps:

1. The list of training images is randomized
2. Run histogram equalization (a technique for adjusting the contrast of images) on all training images.
3. Resize all training images to a 227x227 format
4. The images are divided into a training set and validation set
5. Store the training and validation in 2 LMDB databases. train_lmdb for training the model and validation_lmbd for model evaluation.

Implementation

The implementation process is split into two main stages:

1- Model training
2- Model prediction

After defining the model and the solver, we started training the model .

During the training process, we monitored the loss and the model accuracy. We take a snapshot of the trained model every 5000 iterations, and store them under *caffe_model* folder. The snapshots have *.caffemodel* extension. For example, 5000 iterations snapshot will be called: *caffe_model_1_iter_5000.caffemodel*.
A screenshot of our model break in 3 images for better visualization.

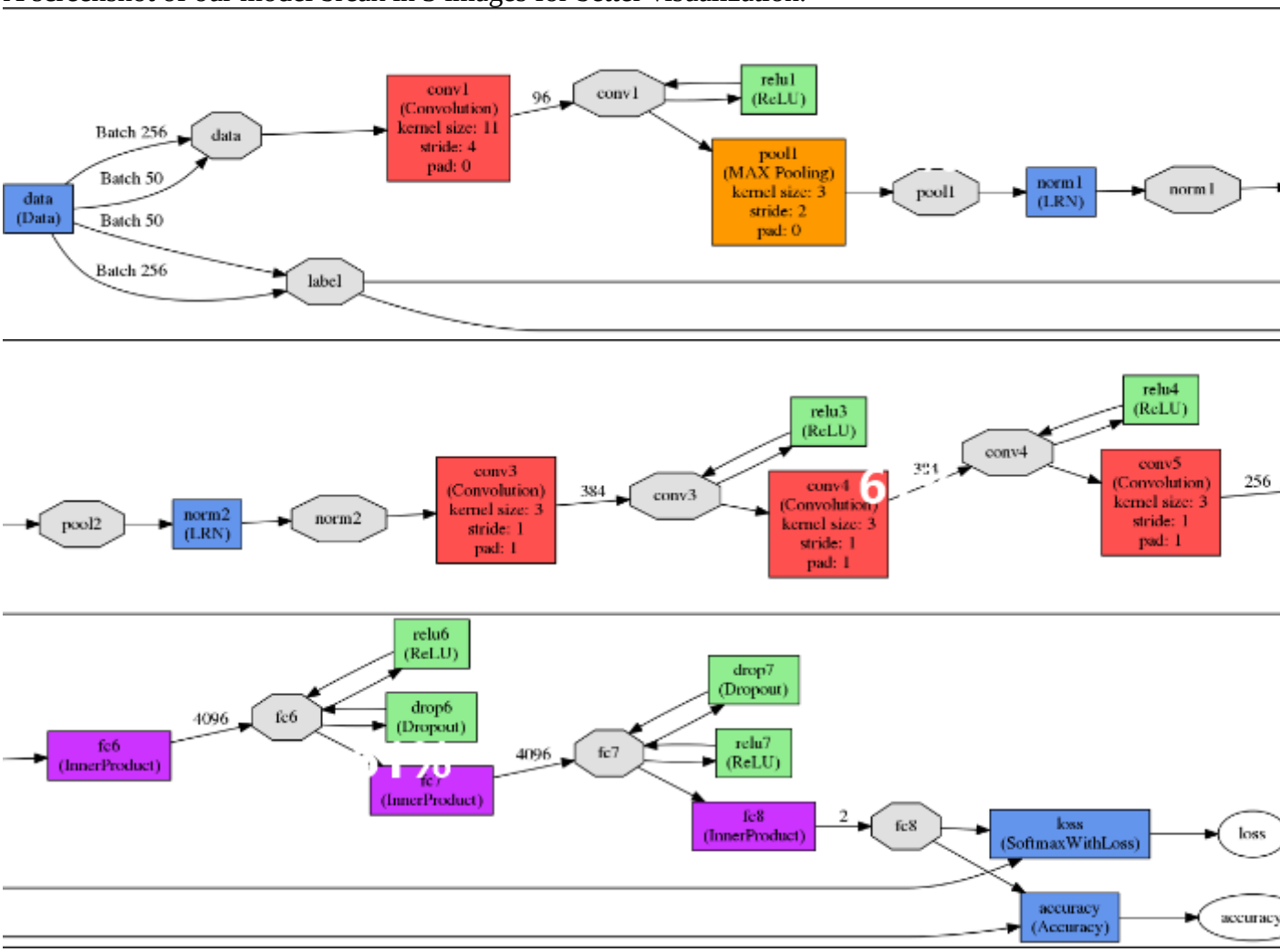

Figure 9: model Architecture image

Training requires:

- *Caffenet_train_val_1.prototxt*: defines the network architecture(figure 9), initialization parameters, and local learning rates

- *Solver_1.prototxt*: defines optimization/training parameters and serves as the actual file that is called to train a deep network. The model we used is **Stochastic gradient descent** (`type: "SGD"`).

  Some important parameters as described in Algorithm and techniques section are described as follow:

```
net: "/home/ubuntu/cats-dogs/caffe_model/caffenet_train_val_1.prototxt"
test_iter: 1000
test_interval: 1000
base_lr: 0.001  # begin training at a learning rate of 0.01 = 1e-2
lr_policy: "step" # learning rate policy: drop the learning rate in "steps"
                  # by a factor of gamma every stepsize iterations
gamma: 0.1 # drop the learning rate by a factor of 10
                  # (i.e., multiply it by a factor of gamma = 0.1)
stepsize: 2500 # drop the learning rate every 2500 iterations
display: 50
max_iter: 40000 # train for 40000 iterations total
momentum: 0.9
weight_decay: 0.0005
snapshot: 5000
snapshot_prefix: "/home/ubuntu/cats-dogs/caffe_model"
solver_mode: GPU
```

  - *Caffenet_deploy_1.prototxt*: used only in testing. It must be exactly the same as *train_val.prototxtexcept* from the input layer(s), loss layer(s), and weights initialization (e.,g weight_filler) as the latter two do not exist in caffe_deploy_1.prototxt.

A learning curve (**Figure 9**) is a plot of the training and test losses as a function of the number of iterations. These plots are very useful to visualize the train/validation losses and validation accuracy.
We can see from the learning curve that the model achieved a validation accuracy of 90%, and it stopped improving after 3000 iterations.

The model prediction stage use the trained model to make predictions on images from test1. The code uses 4 files to run:

a) Test images: We have used test1 images.
b) Mean image: The mean image.

c) Model architecture file: We have called this file *caffenet_deploy_1.prototxt*. It's stored under /caffe_model folder. It's structured in a similar way to caffenet_train_val_1.prototxt, but with a few modifications. We have deleted the data layers, add an input layer and change the last layer type from SoftmaxWithLoss to Softmax.
d) Trained model weights: This is the file that we computed in the training phase. We use caffe_model_1_iter_10000.caffemodel.

The predictions will be stored under caffe_model/submission_model_1.csv.

Refinement

We use a solver for model optimization. We defined the solver's parameters in a .prototxt file. The solver is saved under /caffe_model with name solver_1.prototxt. Below is a copy of the same.

This solver computes the accuracy of the model using the validation set every 1000 iterations. The optimization process will run for a maximum of 40000 iterations and will take a snapshot of the trained model every 5000 iterations.

base_lr, *lr_policy*, *gamma*, *momentum* and *weight_decay* are hyperparameters that we need to tune to get a good convergence of the model. It is described in the previous section.

I chose:

lr_policy: "step" with

stepsize: 2500,

base_lr: 0.001 and

gamma: 0.1.

In this configuration, we started with a learning rate of 0.001, and we dropped the learning rate by a factor of ten every 2500 iterations.

The parameters we used for base_lr, gamma, momentum are does recommended by caffe strategy used by Krizhevsky et al . [1] in their famously winning CNN entry to the ILSVRC-2012 competition. A good strategy for deep learning with SGD is to initialize the learning rate α to a value around $\alpha \approx 0.01 = 10-2$, and dropping it by a constant factor gamma(e.g., 0.1) throughout training when the loss begins to reach an apparent “plateau”, repeating this several times. Generally, it is probably want to use a momentum $\mu = 0.9$ or similar value. By smoothing the weight updates across iterations, momentum tends to make deep learning with SGD both stabler and faster. We have chosen the stepsize = 2,500 instead of 100,000 because of the proportion we made between the size of our training dataset(25,000 images) and the size of Alex Krizhervsky training data. Alex Krizhevsky et al trained a large, deep convolutional neural network to classify the 1.2 million high-resolution images in the ImageNet LSVRC-2010 contest into the 1000 different classes with a stepwise of 100,000.

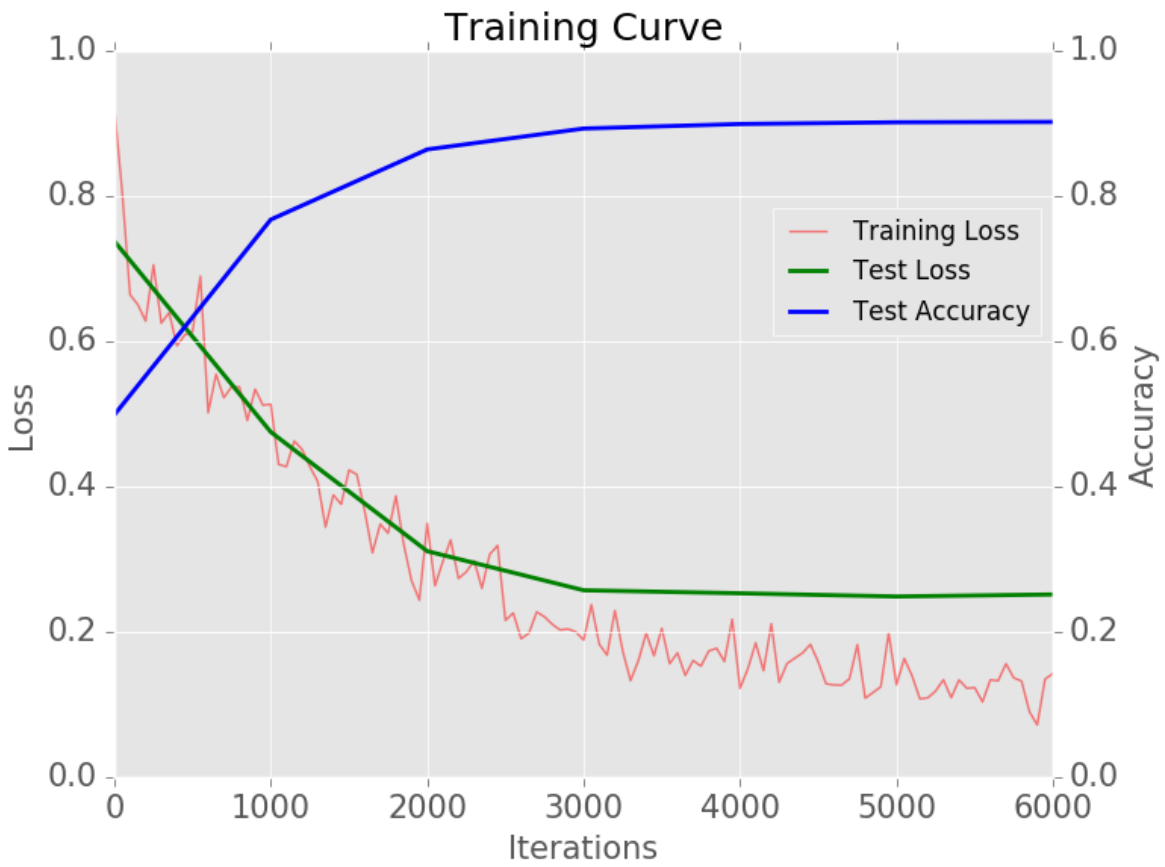

Fig. 10 A plot of the training/validation losses.

After submitting the prediction to Kaggle, it gives an accuracy of 0.89691.

Results

Model Evaluation and validation

During the training step, we divided the training data into training and validation. We use a validation set to evaluate the model. We refer to figure 9 describes to describe the model and the training process, as following:

We can see from the learning curve that the model achieved a validation accuracy of 90%, an d it stopped improving after 3000 iterations.

Justification

The result obtained (0.89691) is less accurate than the benchmark (0.97223). The benchmark uses the Transfer Learning technique.

Instead of training the network from scratch as we have explained previously., transfer learning utilizes a trained model on a different dataset, and adapts it to the problem that we're trying to solve.

There are 2 strategies for transfer learning:

- Utilize the trained model as a fixed feature extractor: In this strategy, we remove the last fully connected layer from the trained model, we freeze the weights of the remaining layers, and we train a machine learning classifier on the output of the remaining layers.
- Fine-tune the trained model: In this strategy, we fine tune the trained model on the new dataset by continuing the backpropagation

Conclusion

Free-form Visualization

No form to visualize.

For this section, we plot two images from the validation set and their respective labels and predictions. Then we will discuss why some of them were incorrectly predicted based on the characteristics of the images.

Label is cat but the model predicts 1(dog)

The model fails in prediction of the previous image because this image is just presenting the back of the animal.

Label is cat but the model predicts 1(dog)

The model fails in prediction of the previous image because this image does not present the visage of the animal, it just presents the feet of the animal. The model does not have enough information to proceed

Reflection

My first challenge was to familiarize with the framework caffe and aws ec2 instance for deep learning, both of which were technologies that I was not familiar with before the project.

Using one AWS EC2 instance of type g2.2xlarge. This instance has a high-performance NVIDIA GPU with 1,536 CUDA cores and 4GB of video memory, 15GB of RAM and 8 vCPUs. The machine costs $0.65/hour with the deep learning framework caffe and anaconda2 installed. This help me to win in time.

Improvement

To improve the model, we can make the following modification:

- Reduces the training time and the size of dataset by using techniques such as Transfer Learning instead of training the model from scratch.
- Add a web interface or mobile interface for our application for the public to use it.

Quality

Presentation

Functionality

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

http://caffe.berkeleyvision.org/tutorial/solver.html

https://en.wikipedia.org/wiki/Convolutional_neural_network

https://en.wikipedia.org/wiki/Lightning_Memory-Mapped_Database

https://en.wikipedia.org/wiki/Evaluation_of_binary_classifiers