|  |  |
| --- | --- |
| **Capstone Project**  **Machine Learning Engineer Nanodegree** | Priyanka Dwivedi  September 2016 |

**Definition**

**Project Overview**

The project I chose is an active Kaggle competition called Dogs vs Cats.

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

The goal of the project is to use supervised learning to be able to distinguish images of dogs from cats. The training dataset for this project is a set of 25000 images that have been labelled as either a cat or a dog. The dataset available on Kaggle has 12,500 images for cats and 12,500 images for dogs. The images are very real life with photos having varying degree of lightening, other objects/people in the dataset along with cats or dogs and the animals are not always centered in the image.

The problem domain for this exercise is **computer vision** and **deep learning**. I am excited about this project as it gives me the chance to explore a field that is up and coming and out of my comfort zone. My background is in consumer finance and this gives me a chance to learn something new!

To succeed in this project I have leveraged several sources to build my understanding of Deep learning and also learn about implementation using TensorFlow. These sources include:

* Udacity Course: [Deep Learning](https://www.google.com/url?q=https://www.udacity.com/course/deep-learning--ud730&sa=D&ust=1473187237687000&usg=AFQjCNEh1l5QuBAKB9jBxl7CM_kpkv_tpw) by Vincent Vanhoucke (Google and Udacity).
* TensorFlow and TFLearn

<http://tflearn.org/getting_started>

<https://www.tensorflow.org/versions/r0.10/tutorials/index.html>

* Books on Deep Learning
  + Artificial Intelligence for Humans, Volume 3: Deep Learning and Neural Networks by Jeff Heaton
  + Getting started with TensorFlow by Giancarlo Zaccone
* [CS231n: Convolutional Neural Networks for Visual Recognition](https://cs231n.github.io/) — Andrej Karpathy's Stanford CS class
* Research papers on latest techniques in convolution neural networks
  + http://arxiv.org/pdf/1412.6806.pdf

**Problem Statement**

The problem here is to use Supervised Learning to train a computer to distinguish between images of dogs and cats. We have a dataset of 25000 images for cats and dogs with labels to accomplish this. Once the computer is trained, we have a test dataset of 12,500 mixed images for which a label (cat or dog) needs to be provided. The success of the project will depend on our accuracy in classifying cat or dog images in the test dataset.

This problem is a classic case of a computer vision challenge. The images are not centered and there are often other objects on the image. Due to variation in scale, rotation and noise between images, my approach would be to train a **deep convolution neural network (CNN)** implemented using TensorFlow to teach the computer to analyze various aspects of the image. My goal is to start with a simple classic implementation of CNN with several layers of convolution and max pooling. And gradually improve on this basic model by adding dropout, regularization and learning rate decay as well as optimizing hyper parameters and training parameters.

One challenge that I will likely encounter is processing power. I don’t have a GPU computer and I am not sure if my Mac with CPU will have enough processing power. If I do need access to GPU, then I will leverage Amazon Web Services, Elastic Cloud 2 GPU for training the model. This will be new for me and another learning experience. The challenge here is to approach the problem in a computationally efficient manner

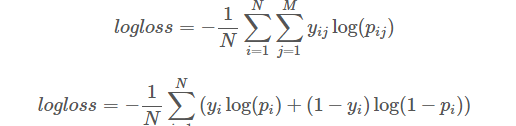
**Metrics**

For this project we have two datasets:

* Train dataset of 25,000 images with labels provided for each image
* Test dataset of 12,500 images. **No labels provided**. After the model has been trained, the test set will be scored on the best model and the output submitted to Kaggle. Kaggle will provide a log loss estimate on the submission

My plan is to divide the train dataset into 3 different sets – 1.Ttraining dataset of 20,000 images that will be used to train the model, 2. Validation dataset of 4,000 images for testing the performance of the CNN and 3. A test set of 1000 images that will be used to do a final evaluation before submission. The goal of creating this test set is that as the CNN is repeatedly trained and tested on validation set, it will eventually “see” the validation set. This test set will be a more objective evaluation of the accuracy of CNN on an out of sample. I will be using two metrics for measuring the success of the project:

1. Log Loss function – The goal of the convolution optimizer will be to minimum the cross entropy loss which is the same as log loss. The selected model would be the one with the lowest log loss on the validation and test sets. This is also the metric that will be used by Kaggle to rank contestants in this competition. The calculation of log loss is below:



Log loss will be computed on training and validation sets before submission. The Log loss for testing dataset will be provided by Kaggle.

2. Accuracy Score – Defined as % labels correctly classified when comparing model prediction vs actual. This metric will be computed on the training dataset and the validation set which is the population for which we have labels for the images. This is the secondary metric that will be computed and shared but the select of the optimal model will be based on minimizing log loss. It is expected that the model with the lowest log loss will also have one of the best accuracy score.

**Analysis**

**Data Exploration and Exploratory Visualization**

The data for this project can be found on Kaggle website and is a zip file of 25,000 images for training for which cat or dog label has been provided and a zip file of 12,500 images for testing with no labels for submission to the competition.

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

Each image in the training has the format “cat.250.jpg” which includes a label for cat or dog and an image sequence number. The data includes real images of cats and dogs. A few images are shared below:

Cats

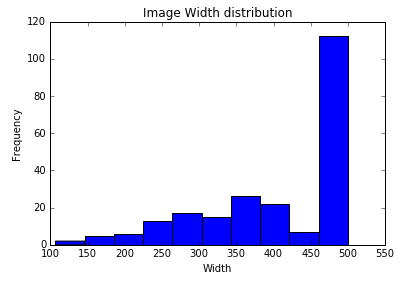
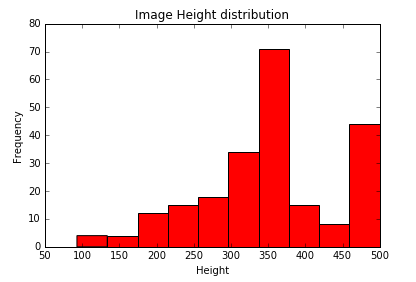
Dogs

Here are a few characteristics of the dataset that make this classification exercise challenging:

1. Presence of people and other objects in the image
2. Some images have multiple animals
3. Images are shot under different amount of light
4. Some images are blurry
5. Animals are not always centered in the image
6. Varying amount of color in the images
7. Images have different height and width

The histogram below takes a random sample of 225 images from the dataset and plots image width and height

We can observe that images have different width and heights. Most pictures have width around 500 pixels and height varies b/w 300-500 pixels.

I think this is a first challenge to solve for. I would like to

1. Have each picture be reduced to a uniform width and height
2. The images are big. To deal with processing efficiently, images should be smaller. Each pixel will be an input into a neural network so large images would require substantial computing power. Example each 56x56 gray image would be convereted to an array of 56x56x1 pixels = 3136 pixels. If instead we go with a colored image of 128x128 then it will have 128x128x3 = 49152 pixels i.e each image will have a dataset that’s 15 times bigger. Multiplying that with 25k images and we can see how this can quickly become a massive dataset
3. I would like to start with a dataset of all 25000 images resized to 56x56 pixels and reduced to grayscale.

**Algorithms and Techniques**

Given the nature of this problem – images with difference in scale, rotation and noise, I think a deep convoluted neural network is the best algorithm to choose for this problem.   Convolutional neural networks are biologically inspired variants of multilayer perceptrons (MLP), designed to emulate the behaviour of a visual cortex. These models mitigate the challenges posed by the MLP architecture by exploiting the strong spatially local correlation present in natural images Also all neurons in a given convolutional layer share the same weights and detect exactly the same thing. Replicating units in this way allows for features to be detected regardless of their position in the visual field, thus constituting the property of translation invariance. Also sharing weights between neurons makes it computationally efficient.

Here is the approach I have in mind:

1. Start with a basic convolution neural network structure – A Convolution Layer with RELU activation function followed by a max pool layer. Multiple such convolution layers will be connected. Finally have one fully connected layer before the output layer. Stochastic gradient descent will be used to optimize this model.

Input Layer -> Conv Layer 1 with RELU -> Max Pool Layer 1 -> Conv Layer 2 with RELU -> Max Pool Layer 2 -> Fully Connected Layer with RELU -> Output Layer

Add regularization to the basic convolution neural network defined in step 1 above to solve for the problem of over fitting. The regularization techniques I will use are dropout and L2 regularization. I would also like to explore learning rate decay to further tune stochastic gradient descent optimization

1. Once a basic convolution neural network is set up, then I will move to the important task of tuning it to the dataset to improve on the results. The following will be done to further tune the basic model:
   1. Change in layer structure of the neural network – Adding more convolution and fully connected layers, replacing max pool layer with a convolution layer etc.
   2. Tuning the no of neurons in the all the different layers
   3. Optimization of regularization techniques – L2 regularization on just fully connected layer vs all layers
   4. Selecting the best optimization function – SGD, Adam etc
   5. Tuning the learning rate in the optimization function
   6. Image augmentation
   7. Increasing no of training epochs
   8. Using color images instead of grayscale images

**Benchmark**

Kaggle has defined a loss rate of **0.5** as the benchmark performance of this competition. Once the model is trained, I will submit my predictions on the test dataset and share the performance on the test dataset with Udacity.

**Methodology**

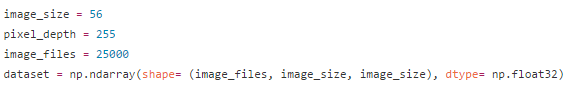
**Data Preprocessing**

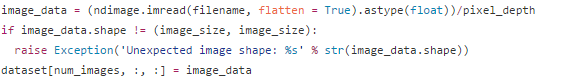
Here are the image processing steps that were undertaken:

1. Resize the images to similar width and height and reduce the size. All images were resized to 56x56 pixels. PIL Image library was used to do the resize and save images as Jpeg. Sample code below:

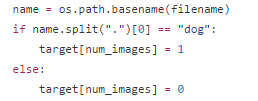


2. Use scipy ndimage to read the image files as a dataset. To begin with all images were read as grayscale images (Flatten = True). Also the resulting dataset was divided by pixel depth = 255 to scale it b/w 0 -1. Sample code below:

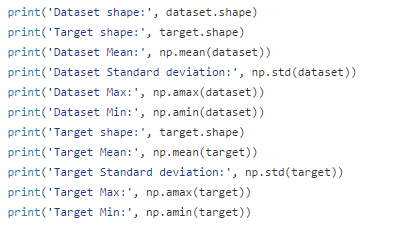




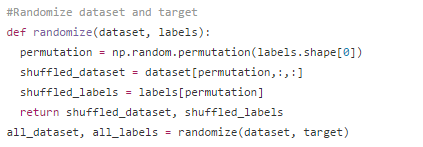
4. Create a dataset for labels. The Kaggle dataset has a jpeg labelled as either cat or dog. The target was extracted from the file name



At this stage sample statistics were calculated on the dataset and target set including shape, mean, standard deviation, min and max. See code below:



5. Shuffle the cat and dog images within the dataset since the pictures were read as first all cat pics and then all dog pics

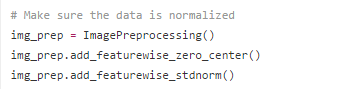


6. Use sklearn cross validation to split the overall dataset into 3 parts – 1. Train set of 20k images, 2. Validation set of 4k images and 3. Testing set of 1k images. This was done in two steps:





7. Before feeding this dataset into the convolution neural network. The dataset was normalized by mean and standard deviation evaluated over all the samples. This was implemented using TFLearn preprocessing module. See sample code below;



8. Reshape the dataset and labels into the proper format before feeding into Tensorflow. The dataset was reshape as a 4D tensor and the labels as two columns – one for cat and another for dog. See sample code below:





No further pre-processing was done for images since convolution neural networks work best when input images are kept close to natural state and the model is allowed to extract all relevant features. I did explore adding ZCA whitening as a preprocessing step however that reduced the log loss of the final model hence dropped. The full code for preprocessing can be found on my GitHub repository:

<https://github.com/priya-dwivedi/Udacity-ML-Nano-Degree/blob/master/P5/create_new_pickle_4.py>

<https://github.com/priya-dwivedi/Udacity-ML-Nano-Degree/blob/master/P5/catvsdog_vanilla.py>

**Implementation**

I started with implementing a simple convolution neural network in TFLearn. This neural network had the following structure

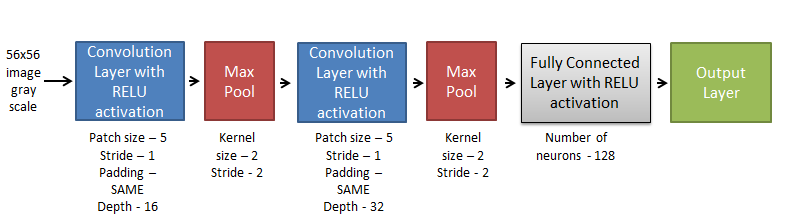


Figure 1: Simple Convolution Neural Network

The algorithm was created using TFLearn tutorial on their website

<http://tflearn.org/getting_started/>

Batch size of 96 neurons was used to train the model. I found the model very slow to train on my Mac CPU so I used Amazon Web Service Elastic Cloud 2 GPU 2.2x Large to train this model. After experimenting with several parameters and running the model for 15 epochs, I got the following results:

**Best Validation Accuracy Score: 78.78%**

**Valid Log Loss: 0.5487**

**Test Log Loss: 0.5306**

The full code for this model is shared at my GitHub repository:

<https://github.com/priya-dwivedi/Udacity-ML-Nano-Degree/blob/master/P5/catvsdog_vanilla.py>

Sample code is shared below:



**Refinement**

Bulk of my time in this project was spent on this stage – tuning of hyper parameters and training parameters. This section summarizes all the different steps I took and the results for the same. Use of Amazon EC2 GPU really helped at this stage in being able to experiment with different parameters efficiently. The following adjustments were tried to further refine the model:

1. Change in layer structure of the neural network – Adding more convolution and fully connected layers, placement of max pool layer and one setting with replacing max pool layer with convolution layer of same stride
2. Tuning of neural network parameters – patch size and depth of convolution layers and number of neurons in the fully connected layer
3. Optimization of regularization techniques – L2 regularization on just fully connected layer vs all layers
4. Selecting the best optimization function – Stochastic Gradient Descent or Adam Optimizer
5. Tuning the learning rate in the optimization function
6. Image augmentation – Use of rotation, blur, flip left to right and crop to augment the training set
7. Increasing no of training epochs
8. Using color images instead of grayscale images

The above techniques were selected as I felt that they would influence the model output. I used my learning from the books, courses and research papers that I read for this project to select this list. The table below summarizes the key adjustments made and results of the same.

You can also see the excel file for this at my GitHub repository:

<https://github.com/priya-dwivedi/Udacity-ML-Nano-Degree/blob/master/P5/refinement_results.xlsx>

Table 1 – Results from refinement of model



The table covers most of the important adjustments to the model. The green cell highlights what was changed in that refinement from the row above. The 3 columns on the right summarize the results of the changes on the log loss and accuracy of the model. Please note the shorthand: conv – Convolution Layer, MP – Max Pool Layer and FC – Fully connected layer.

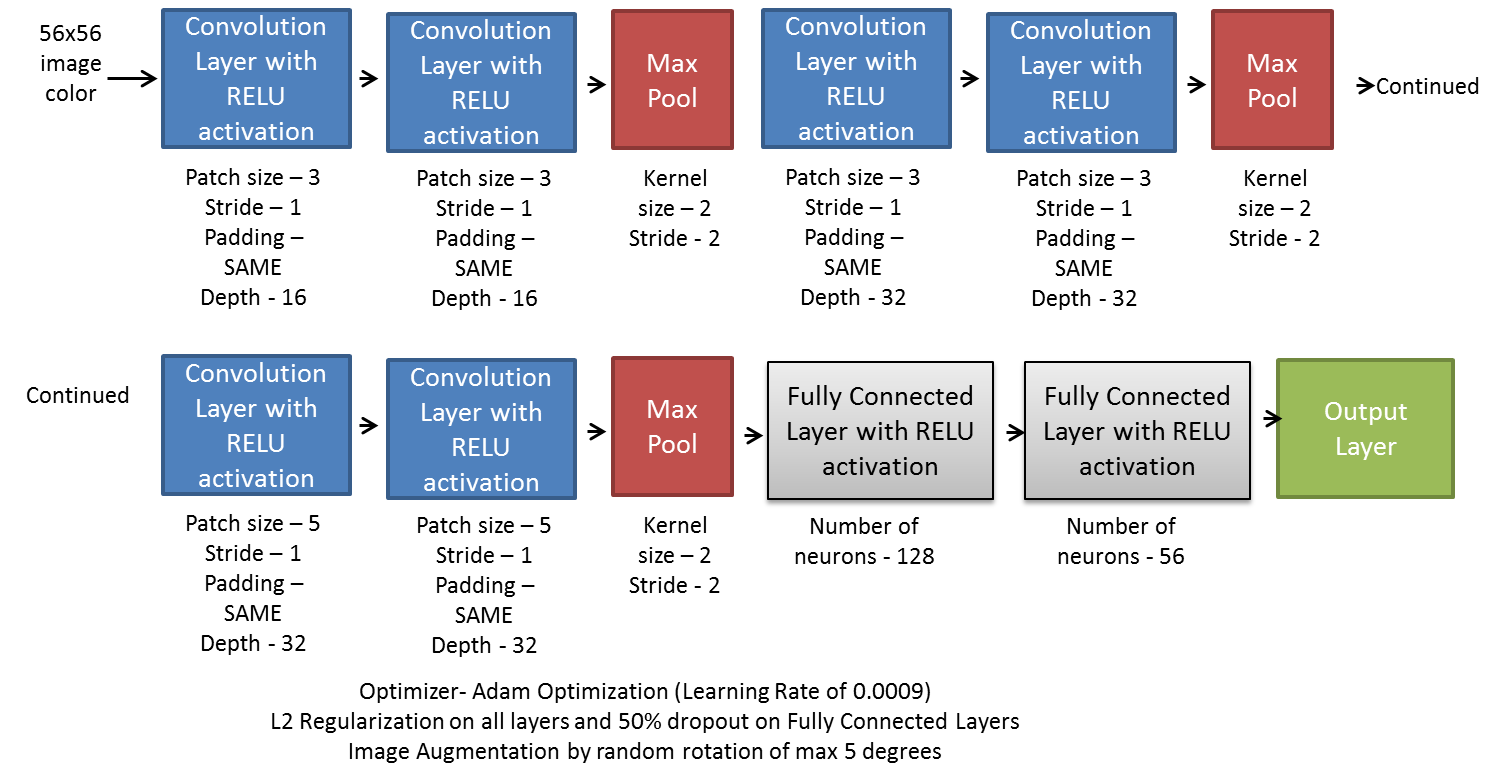
The initial model is highlighted in the top row of the table and the final model was the refinement 16 highlighted in yellow. We can see significant improvements in performance between initial and final model.



**Results**

**Model Evaluation and Validation**

The figure below shows the structure of the final model:



The code for this model is saved on my GitHub repository

<https://github.com/priya-dwivedi/Udacity-ML-Nano-Degree/blob/master/P5/catvsdog_color1.py>

This model was chosen as the final model for several reasons:

1. I tried all the techniques I possibly could to refine the model

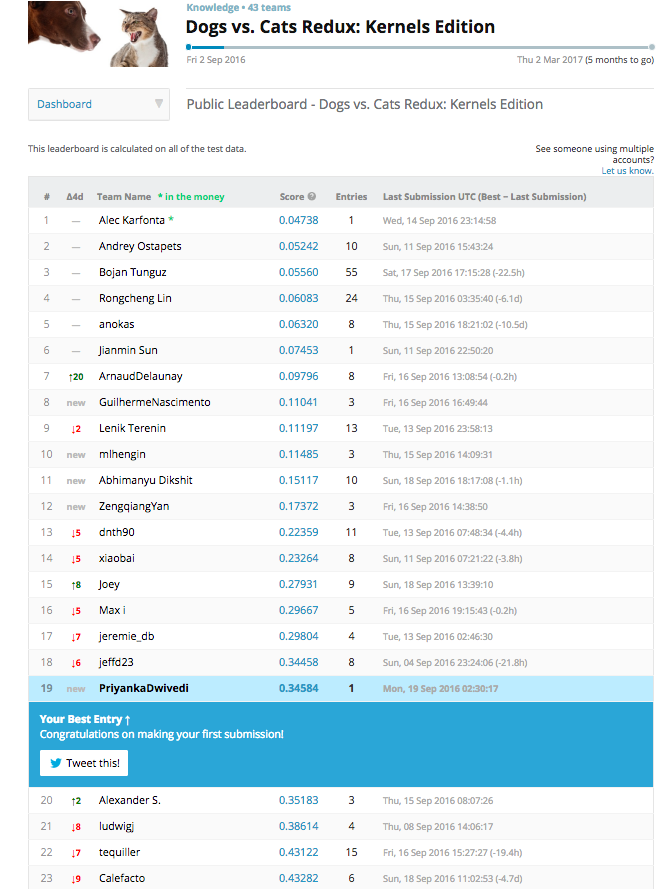
2. As can be seen from table 1, the incremental benefit of changes to the model was very small after a point. So theoretically refinement can continue but in my opinion the effect would be small and not worth the time

3. Log loss is lower than the benchmark established in the beginning (0.5) and the model accuracy score of 85% on validation set is reasonable

To evaluate the robustness of this model on data never seen before, I used the test set on Kaggle website to make a submission. The test set is a dataset of 12.500 images for which a label has not been provided. The trained model was used to score this dataset and the results were submitted to Kaggle. The code for this part is also on my GitHub repository:

<https://github.com/priya-dwivedi/Udacity-ML-Nano-Degree/blob/master/P5/kaggle_test_pred.py>

The Log loss on the test dataset at Kaggle was **0.3458.** This is in the ballpark we had with the final model on our validation and test sample. As expected performance is slightly weaker on the sample never seen before however the model seems robust. The screen shot below shows performance on Kaggle dataset.



You can also see the latest leaderboard on Kaggle to verify the same (ranking would fluctuate over time as the competition is still open).

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

**Justification**

The figure below shows the structure of the final model:

“*Real-world example*. The [Krizhevsky et al.](http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks) architecture that won the ImageNet challenge in 2012 accepted images of size [227x227x3]. On the first Convolutional Layer, it used neurons with receptive field size F=11F=11, stride S=4S=4and no zero padding P=0P=0. Since (227 - 11)/4 + 1 = 55, and since the Conv layer had a depth of K=96K=96, the Conv layer output volume had size [55x55x96]. Each of the 55\*55\*96 neurons in this volume was connected to a region of size [11x11x3] in the input volume. Moreover, all 96 neurons in each depth column are connected to the same [11x11x3] region of the input, but of course with different weights. As a fun aside, if you read the actual paper it claims that the input images were 224x224, which is surely incorrect because (224 - 11)/4 + 1 is quite clearly not an integer. This has confused many people in the history of ConvNets and little is known about what happened. My own best guess is that Alex used zero-padding of 3 extra pixels that he does not mention in the paper.”

INPUT -> [CONV -> RELU -> CONV -> RELU -> POOL]\*3 -> [FC -> RELU]\*2 -> FC Here we see two CONV layers stacked before every POOL layer. This is generally a good idea for larger and deeper networks, because multiple stacked CONV layers can develop more complex features of the input volume before the destructive pooling operation.

*Prefer a stack of small filter CONV to one large receptive field CONV layer*. Suppose that you stack three 3x3 CONV layers on top of each other (with non-linearities in between, of course). In this arrangement, each neuron on the first CONV layer has a 3x3 view of the input volume. A neuron on the second CONV layer has a 3x3 view of the first CONV layer, and hence by extension a 5x5 view of the input volume. Similarly, a neuron on the third CONV layer has a 3x3 view of the 2nd CONV layer, and hence a 7x7 view of the input volume. Suppose that instead of these three layers of 3x3 CONV, we only wanted to use a single CONV layer with 7x7 receptive fields. These neurons would have a receptive field size of the input volume that is identical in spatial extent (7x7), but with several disadvantages. First, the neurons would be computing a linear function over the input, while the three stacks of CONV layers contain non-linearities that make their features more expressive. Second, if we suppose that all the volumes have CC channels, then it can be seen that the single 7x7 CONV layer would contain C×(7×7×C)=49C2C×(7×7×C)=49C2 parameters, while the three 3x3 CONV layers would only contain 3×(C×(3×3×C))=27C23×(C×(3×3×C))=27C2 parameters. Intuitively, stacking CONV layers with tiny filters as opposed to having one CONV layer with big filters allows us to express more powerful features of the input, and with fewer parameters. As a practical disadvantage, we might need more memory to hold all the intermediate CONV layer results if we plan to do backpropagation.

Layer Sizing Patterns

Until now we’ve omitted mentions of common hyperparameters used in each of the layers in a ConvNet. We will first state the common rules of thumb for sizing the architectures and then follow the rules with a discussion of the notation:

The **input layer** (that contains the image) should be divisible by 2 many times. Common numbers include 32 (e.g. CIFAR-10), 64, 96 (e.g. STL-10), or 224 (e.g. common ImageNet ConvNets), 384, and 512.

The **conv layers** should be using small filters (e.g. 3x3 or at most 5x5), using a stride of S=1S=1, and crucially, padding the input volume with zeros in such way that the conv layer does not alter the spatial dimensions of the input. That is, when F=3F=3, then using P=1P=1 will retain the original size of the input. When F=5F=5, P=2P=2. For a general FF, it can be seen that P=(F−1)/2P=(F−1)/2 preserves the input size. If you must use bigger filter sizes (such as 7x7 or so), it is only common to see this on the very first conv layer that is looking at the input image.

The **pool layers** are in charge of downsampling the spatial dimensions of the input. The most common setting is to use max-pooling with 2x2 receptive fields (i.e. F=2F=2), and with a stride of 2 (i.e. S=2S=2). Note that this discards exactly 75% of the activations in an input volume (due to downsampling by 2 in both width and height). Another slightly less common setting is to use 3x3 receptive fields with a stride of 2, but this makes. It is very uncommon to see receptive field sizes for max pooling that are larger than 3 because the pooling is then too lossy and aggressive. This usually leads to worse performance.

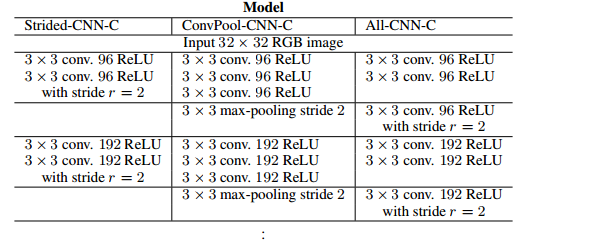
*Reducing sizing headaches.* The scheme presented above is pleasing because all the CONV layers preserve the spatial size of their input, while the POOL layers alone are in charge of down-sampling the volumes spatially. In an alternative scheme where we use strides greater than 1 or don’t zero-pad the input in CONV layers, we would have to very carefully keep track of the input volumes throughout the CNN architecture and make sure that all strides and filters “work out”, and that the ConvNet architecture is nicely and symmetrically wired.

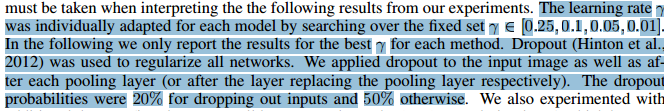
*Why use stride of 1 in CONV?* Smaller strides work better in practice. Additionally, as already mentioned stride 1 allows us to leave all spatial down-sampling to the POOL layers, with the CONV layers only transforming the input volume depth-wise.

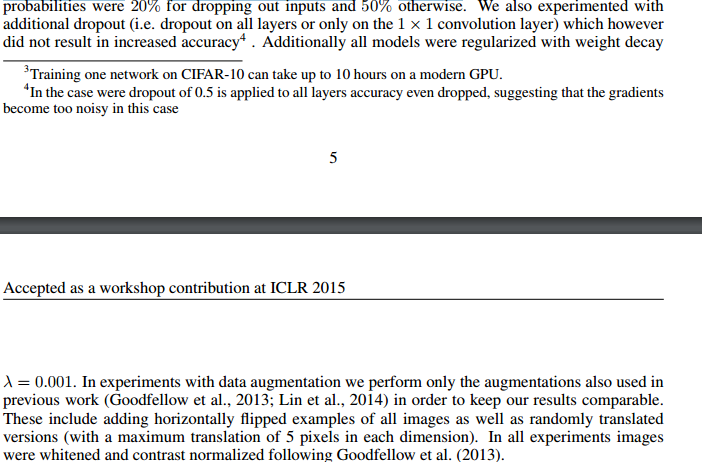
*Why use padding?* In addition to the aforementioned benefit of keeping the spatial sizes constant after CONV, doing this actually improves performance. If the CONV layers were to not zero-pad the inputs and only perform valid convolutions, then the size of the volumes would reduce by a small amount after each CONV, and the information at the borders would be “washed away” too quickly.

*Compromising based on memory constraints.* In some cases (especially early in the ConvNet architectures), the amount of memory can build up very quickly with the rules of thumb presented above. For example, filtering a 224x224x3 image with three 3x3 CONV layers with 64 filters each and padding 1 would create three activation volumes of size [224x224x64]. This amounts to a total of about 10 million activations, or 72MB of memory (per image, for both activations and gradients). Since GPUs are often bottlenecked by memory, it may be necessary to compromise. In practice, people prefer to make the compromise at only the first CONV layer of the network. For example, one compromise might be to use a first CONV layer with filter sizes of 7x7 and stride of 2 (as seen in a ZF net). As another example, an AlexNet uses filer sizes of 11x11 and stride of 4.

<http://arxiv.org/pdf/1412.6806.pdf>







2. Log Loss calculation

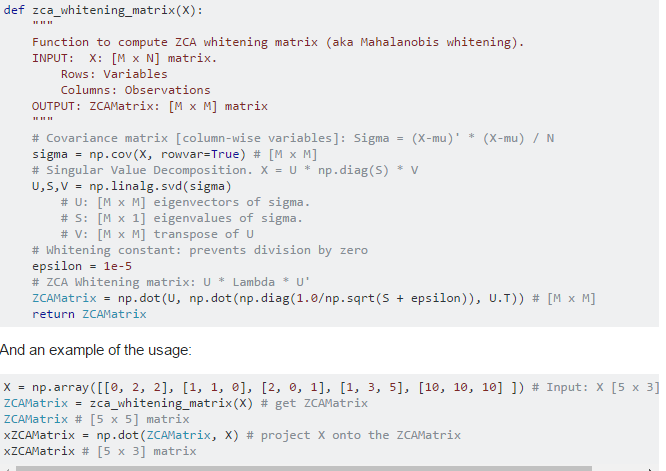
3. saving tensorflow models

4. Image augmentation

5. Image pre processing – ZCA whitening and global contrast normalization.

| **Description** |
| --- |
| Model | Very Deep Convolutional Networks with 3x3 kernel [1] |
| Data Augmentation | cropping, horizontal reflection [2] and scaling. see lib/data\_augmentation.lua |
| Preprocessing | Global Contrast Normalization (GCN) and ZCA whitening. see lib/preprocessing.lua |

<http://www.cs.toronto.edu/~adeandrade/assets/bpfcnnatorii.pdf>



6. Not inputting validation model as a tf constant

Tensorflow documentation has the same

**Learning Rate**

The learning rate is one of, if not the most important hyperparameter. If this is too large or too small, your network may learn very poorly, very slowly, or not at all. Typical values for the learning rate are in the range of 0.1 to 1e-6, though the optimal learning rate is usually data (and network architecture) specific. Some simple advice is to start by trying three different learning rates – 1e-1, 1e-3, and 1e-6 – to get a rough idea of what it should be, before further tuning this. Ideally, they run models with different learning rates simultaneously to save time.