**PART 1 - Image recognition with TensorFlow and Keras**

**Use computer vision, TensorFlow, and Keras for image classification and processing**

By [Prashant Sharma](https://developer.ibm.com/profiles/prashsh1)   
Updated February 19, 2019 | Published January 30, 2019

<https://developer.ibm.com/articles/image-recognition-challenge-with-tensorflow-and-keras-pt1/>

Deep neural networks and deep learning have become popular in past few years, thanks to the breakthroughs in research, starting from [AlexNet](http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks), [VGG](https://arxiv.org/abs/1409.1556), [GoogleNet](https://arxiv.org/abs/1409.4842), and [ResNet](https://arxiv.org/abs/1512.03385). In 2015, with ResNet, the performance of large-scale image recognition saw a huge improvement in accuracy and helped increase the popularity of deep neural networks.

This article discusses using a basic deep neural network to solve an image recognition problem. Here, emphasis is more on the overall technique and use of a library than perfecting the model. [Part 2](https://developer.ibm.com/articles/image-recognition-challenge-with-tensorflow-and-keras-pt2) explains how to improve the results.

I wanted to use a deep neural network to solve something other than a “hello world” version of image recognition — MNIST handwritten letter recognition, for example. After going through the first tutorial on the TensorFlow and Keras libraries, I began with the challenge of classifying whether a given image is a chihuahua (a dog breed) or a muffin from a set of images that look similar.

The data set included with this article is formed by combining this [source](https://github.com/ieee8023/deep-learning-datasets/tree/master/chihuahua-muffin) and searching the internet and applying some basic image processing techniques. The images in this data set are collected, used, and provided under the [Creative commons fair usage policy](https://guides.library.stonybrook.edu/copyright). The intended use is (for scientific research in image recognition using artificial neural networks) by using the TensorFlow and Keras library. This solution applies the same techniques as given in <https://www.tensorflow.org/tutorials/keras/basic_classification>.

Basically, there are no prerequisites to this article, but if you want to follow the code, it’s helpful to have basic knowledge of Python, numpy, and going through th eTensorFlow and Keras library.



**Import the data**

**Clone the Git repository**

$ git clone https://github.com/ScrapCodes/image-recognition-tensorflow.git

$ cd image-recognition-tensorflow

$ python

>>>

**Import TensorFlow, Keras, and other helper libraries**

I used TensorFlow and Keras for running the machine learning and the Pillow Python library for image processing.

Using pip, these can be installed on macOS as follows:

sudo pip install tensorflow matplotlib pillow

*Note: Whether the use of sudo is required depends on how Python and pip is installed on your system. Systems configured with a virtual environment might not need sudo.*

Importing the Python libraries.

# TensorFlow and tf.keras

import tensorflow as tf

from tensorflow import keras

# Helper libraries

import numpy as np

import matplotlib.pyplot as plt

import glob, os

import re

# Pillow

import PIL

from PIL import Image

**Load the data**

A Python function to preprocess input images. For images to be converted into numpy arrays, they must have same dimensions:

# Use Pillow library to convert an input jpeg to a 8 bit grey scale image array for processing.

def jpeg\_to\_8\_bit\_greyscale(path, maxsize):

img = Image.open(path).convert('L') # convert image to 8-bit grayscale

# Make aspect ratio as 1:1, by applying image crop.

# Please note, croping works for this data set, but in general one

# needs to locate the subject and then crop or scale accordingly.

WIDTH, HEIGHT = img.size

if WIDTH != HEIGHT:

m\_min\_d = min(WIDTH, HEIGHT)

img = img.crop((0, 0, m\_min\_d, m\_min\_d))

# Scale the image to the requested maxsize by Anti-alias sampling.

img.thumbnail(maxsize, PIL.Image.ANTIALIAS)

return np.asarray(img)

A Python function to load the data set from images, into numpy arrays:

def load\_image\_dataset(path\_dir, maxsize):

images = []

labels = []

os.chdir(path\_dir)

for file in glob.glob("\*.jpg"):

img = jpeg\_to\_8\_bit\_greyscale(file, maxsize)

if re.match('chihuahua.\*', file):

images.append(img)

labels.append(0)

elif re.match('muffin.\*', file):

images.append(img)

labels.append(1)

return (np.asarray(images), np.asarray(labels))

We should scale the images to some standard size smaller than actual image resolution. These images are more than 170×170, so we scale them all to 100×100 for further processing:

maxsize = 100, 100

To load the data, let’s execute the following functions and load training and test data sets:

(train\_images, train\_labels) = load\_image\_dataset('/Users/yourself/image-recognition-tensorflow/chihuahua-muffin', maxsize)

(test\_images, test\_labels) = load\_image\_dataset('/Users/yourself/image-recognition-tensorflow/chihuahua-muffin/test\_set', maxsize)

* train\_images and train\_lables is training data set.
* test\_images and test\_labels is testing data set for validating the model’s performance against unseen data.

Finally, we define the class names for our data set. Because this data has only two classes (an image can either be a Chihuahua or a Muffin), we have class\_names as follows:

class\_names = ['chihuahua', 'muffin']

**Explore the data**

In this data set, we have 26 training examples, of both Chihuahua and muffin images:

train\_images.shape

(26, 100, 100)

Each image has its respective label – either a 0 or 1. A 0indicates a class\_names[0] i.e. a chihuahua and 1 indicates class\_names[1] i.e. a muffin:

print(train\_labels)

[0 0 0 0 1 1 1 1 1 0 1 0 0 1 1 0 0 1 1 0 1 1 0 1 0 0]

For test set, we have 14 examples, seven for each class:

test\_images.shape

(14, 100, 100)

print(test\_labels)

[0 0 0 0 0 0 0 1 1 1 1 1 1 1]

**Visualize the data set**

Using the matplotlib.pyplot Python library, we can visualize our data. Make sure you have the matplotlib library installed.

Following Python helper function helps us draw these images on our screen:

def display\_images(images, labels):

plt.figure(figsize=(10,10))

grid\_size = min(25, len(images))

for i in range(grid\_size):

plt.subplot(5, 5, i+1)

plt.xticks([])

plt.yticks([])

plt.grid(False)

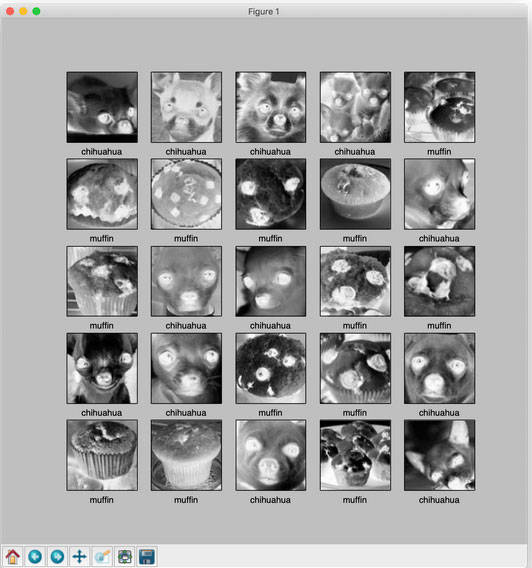
plt.imshow(images[i], cmap=plt.cm.binary)

plt.xlabel(class\_names[labels[i]])

Let’s visualize the training data set, as follows:

display\_images(train\_images, train\_labels)

plt.show()



**Note: The images are grayscaled and cropped in the preprocessing step of our images at the time of loading.**

Similarly, we can visualize our test data set. Both training and test sets are fairly limited, so feel free to use Google search and add more examples and see how things improve or perform.

**Preprocess the data**

**Scaling the images to values between 0 and 1**

train\_images = train\_images / 255.0

test\_images = test\_images / 255.0

**Build the model**

**Set up the layers**

We have used four layers total. The first layer is to simply flatten the data set into a single array and does not get training. The other three layers are dense and use sigmoid as activation function:

# Setting up the layers.

model = keras.Sequential([

keras.layers.Flatten(input\_shape=(100, 100)),

keras.layers.Dense(128, activation=tf.nn.sigmoid),

keras.layers.Dense(16, activation=tf.nn.sigmoid),

keras.layers.Dense(2, activation=tf.nn.softmax)

])

**Compile the model**

The optimizer is stochastic gradient descent (SGD):

sgd = keras.optimizers.SGD(lr=0.01, decay=1e-5, momentum=0.7, nesterov=True)

model.compile(optimizer=sgd,

loss='sparse\_categorical\_crossentropy',

metrics=['accuracy'])

**Train the model**

model.fit(train\_images, train\_labels, epochs=100)

Three training iterations appear:

....

Epoch 98/100

26/26 [==============================] - 0s 555us/step - loss: 0.3859 - acc: 0.9231

Epoch 99/100

26/26 [==============================] - 0s 646us/step - loss: 0.3834 - acc: 0.9231

Epoch 100/100

26/26 [==============================] - 0s 562us/step - loss: 0.3809 - acc: 0.9231

<tensorflow.python.keras.callbacks.History object at 0x11e6c9590>

**Evaluate accuracy**

test\_loss, test\_acc = model.evaluate(test\_images, test\_labels)

print('Test accuracy:', test\_acc)

14/14 [==============================] - 0s 8ms/step

('Test accuracy:', 0.7142857313156128)

Test accuracy is less than training accuracy. This indicates model has overfit the data. There are techniques to overcome this, and we will discuss those later. This model is a good example of the use of API, but far from perfect.

With recent advances in image recognition and using more training data, we can perform much better on this data set challenge.

**Make predictions**

To make predictions, we can simply call predict on the generated model:

predictions = model.predict(test\_images)

print(predictions)

[[0.6080283 0.3919717 ]

[0.5492342 0.4507658 ]

[0.54102856 0.45897144]

[0.6743213 0.3256787 ]

[0.6058993 0.39410067]

[0.472356 0.5276439 ]

[0.7122982 0.28770176]

[0.5260602 0.4739398 ]

[0.6514299 0.3485701 ]

[0.47610506 0.5238949 ]

[0.5501717 0.4498284 ]

[0.41266635 0.5873336 ]

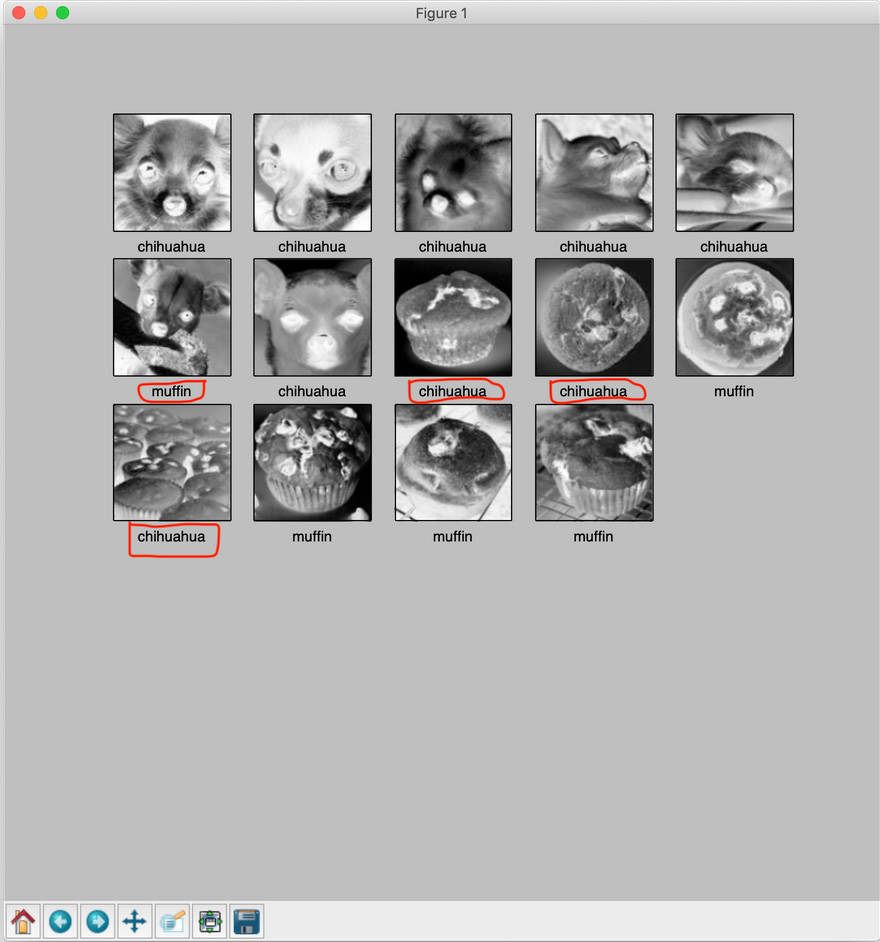
[0.18961382 0.8103862 ]

[0.35493374 0.64506626]]

Finally, display images and see how the model performed on test set:

display\_images(test\_images, np.argmax(predictions, axis = 1))

plt.show()



**Conclusion**

In this article, there are a few wrong classifications in our result, as highlighted in the previous image. So this is far from perfect. In [Part 2](https://developer.ibm.com/articles/image-recognition-challenge-with-tensorflow-and-keras-pt2), we will learn how improve the training.

**PART 2 - Refine your deep learning model**

**Improve your neural network model by using some well-known machine learning techniques**

By [Prashant Sharma](https://developer.ibm.com/profiles/prashsh1)   
Updated February 19, 2019 | Published February 12, 2019

https://developer.ibm.com/articles/image-recognition-challenge-with-tensorflow-and-keras-pt2/

As part of my own learning, continuing from [Part 1](https://developer.ibm.com/articles/image-recognition-challenge-with-tensorflow-and-keras-pt1) and trying to improve our neural network model, we will use some of the well-known machine learning techniques mentioned on [TensorFlow](https://www.tensorflow.org/tutorials/keras/).

In the previous article, we saw certain problems with our training. Here, we will address them and see if our results improve as we go.

**Problems observed in the previous solution**

**Overfitting**

A model is considered to overfit when it performs with great accuracy on the training data (data used for training the model), but when evaluated against a test or unseen data set, it performs rather poorly. This happens because our model has overfit the data.

Training accuracy if higher than testing accuracy is a clear indicator of this phenomenon. Thankfully, there are some techniques available to solve this.

**Model size**

First, look at the size of the model, meaning the number of units. If the model used is far bigger than the problem at hand, it is more likely to learn the features/patterns not relevant to the problem and thus overfit to the training data. A larger model will not generalize well, and a smaller model will underfit the data.

Taking the model in our previous article as a baseline, we will evaluate the result of reducing the size and increasing the size on the performance of the model. The following models were tried and compared:

baseline\_model = keras.models.Sequential([

keras.layers.Flatten(input\_shape = ( maxsize\_w, maxsize\_h , 1)),

keras.layers.Dense(128, activation=tf.nn.sigmoid),

keras.layers.Dense(16, activation=tf.nn.sigmoid),

keras.layers.Dense(2, activation=tf.nn.softmax)

])

bigger\_model2 = keras.models.Sequential([

keras.layers.Flatten(input\_shape = ( maxsize\_w, maxsize\_h , 1)),

keras.layers.Dense(1024, activation=tf.nn.relu),

keras.layers.Dense(512, activation=tf.nn.relu),

keras.layers.Dense(64, activation=tf.nn.relu),

keras.layers.Dense(16, activation=tf.nn.relu),

keras.layers.Dense(2, activation=tf.nn.softmax)

])

bigger\_model1 = keras.models.Sequential([

keras.layers.Flatten(input\_shape = ( maxsize\_w, maxsize\_h , 1)),

keras.layers.Dense(512, activation=tf.nn.relu),

keras.layers.Dense(128, activation=tf.nn.relu),

keras.layers.Dense(16, activation=tf.nn.relu),

keras.layers.Dense(2, activation=tf.nn.softmax)

])

smaller\_model1 = keras.models.Sequential([

keras.layers.Flatten(input\_shape = ( maxsize\_w, maxsize\_h , 1)),

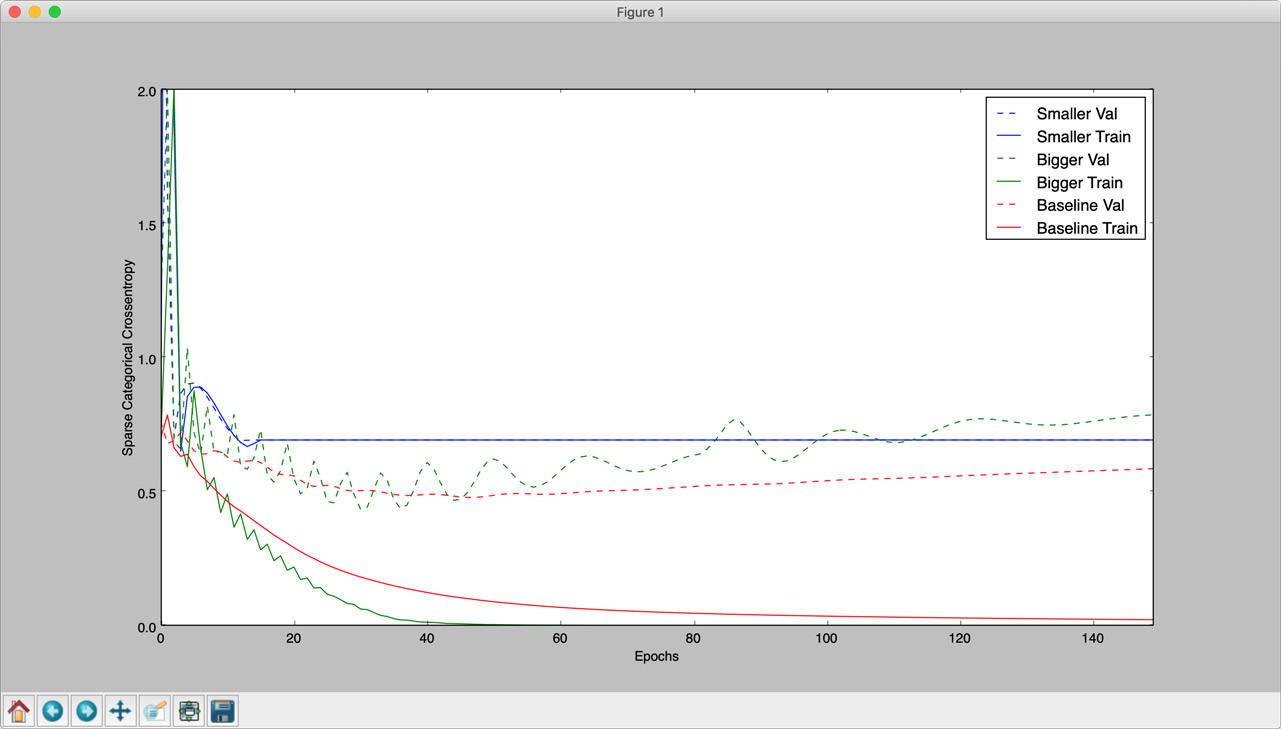
keras.layers.Dense(64, activation=tf.nn.relu),

keras.layers.Dense(2, activation=tf.nn.softmax)

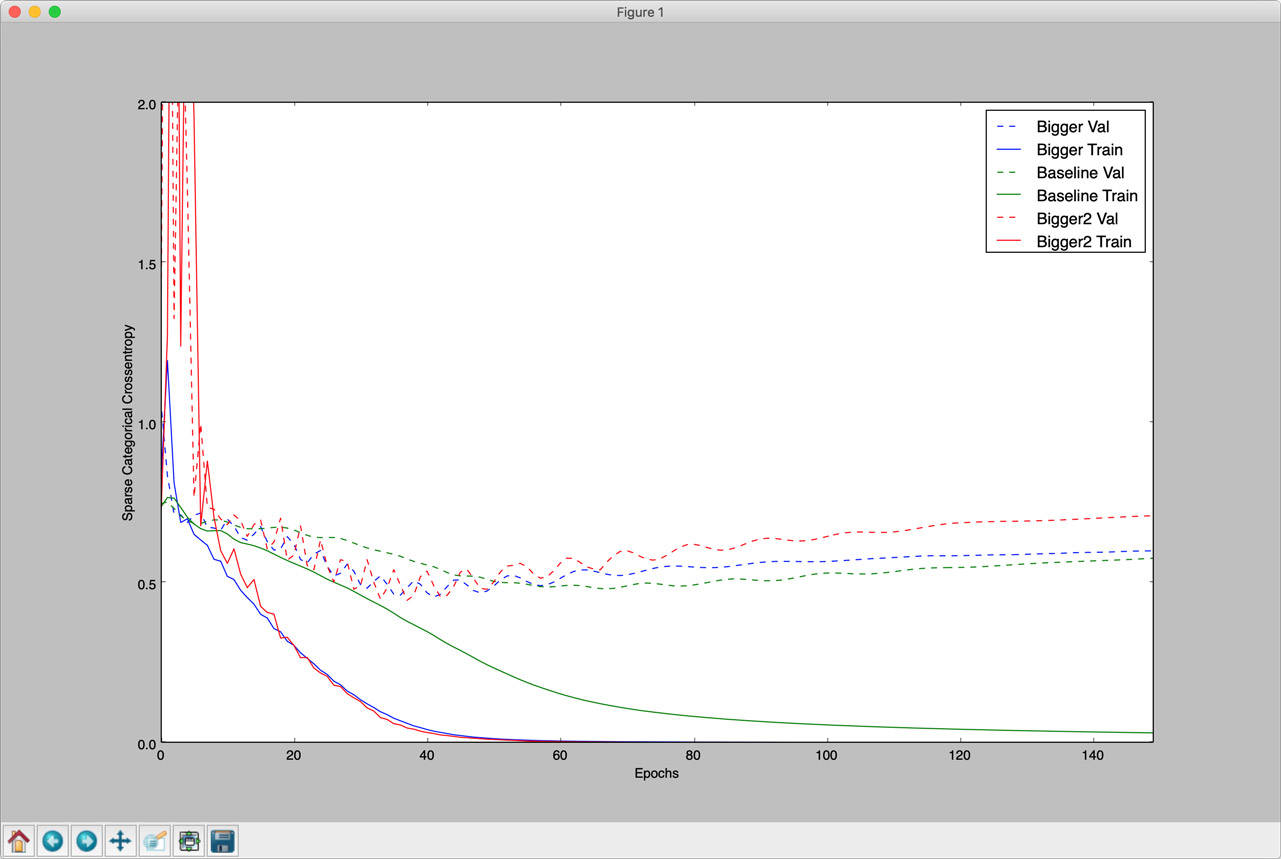
])

To determine the ideal model, we plot loss function of validation data against number of epochs.

1. Comparison of smaller, bigger, and baseline models.



1. Comparison of bigger, bigger2, and baseline models.



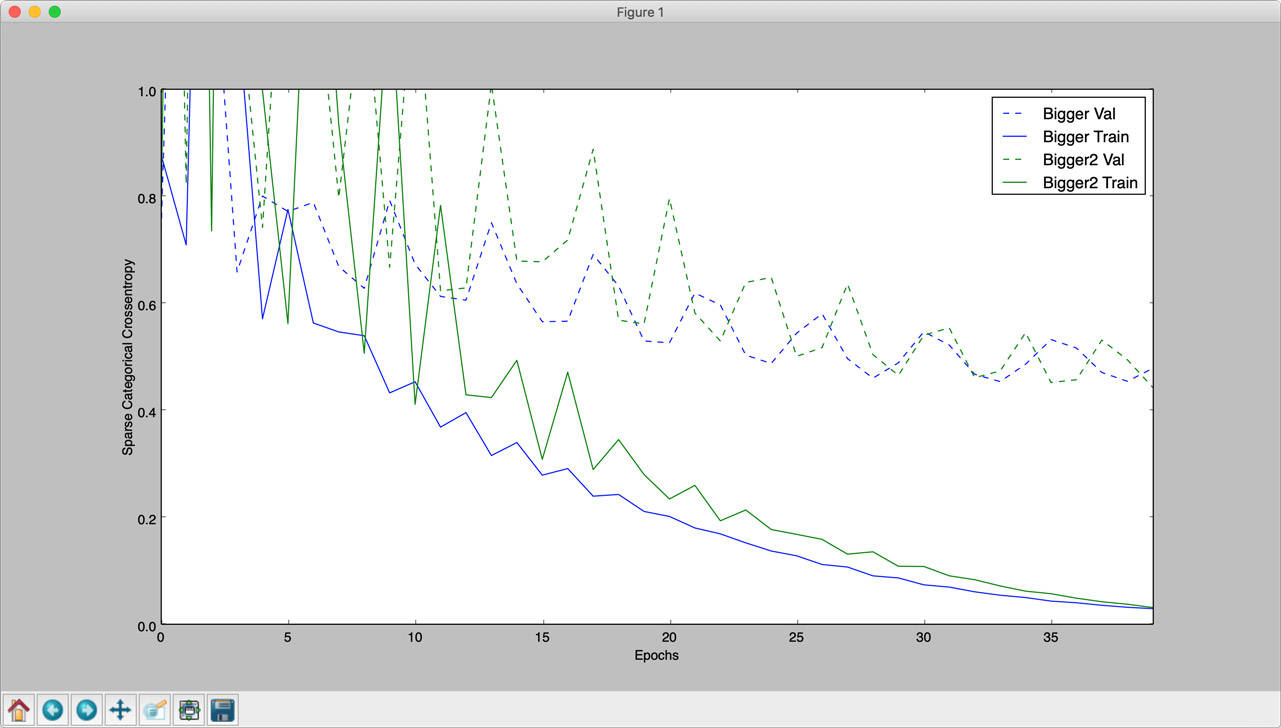
In these plots, we see that validation loss (sparse\_categorical\_crossentropy) is almost similar for bigger and bigger2 models, and better than smaller and baseline models. So we go ahead and select these models over our baseline model for further tuning.

**Number of epochs**

The number of epochs plays an important role in avoiding overfitting and overall model performance. In the comparison graphs plotted in above section, we observe the loss function for validation data reaches a minimum and on further training, increases again, while loss function of training data reduces further. This is exactly what overfitting means; the model learns patterns specific to the data set and does not generalize well, so it does better with training data than validation data. We have to stop before the model overfits the data. So in above case epoch value of 40 is ideal.

**L1 and L2 regularization**

The effect of applying L2 regularization is that of adding some random noise to the layers. The plot below shows the effect of applying this on our model.



**Using Dropout**

Keras library provides a dropout layer, a concept introduced in *Dropout: A Simple Way to Prevent Neural Networks from Overfitting(JMLR 2014)*. A consequence of adding a dropout layer is that training time is increased, and if the dropout is high, underfitting.

Models after applying the dropout layers:

bigger\_model1 = keras.models.Sequential([

keras.layers.Flatten(input\_shape = ( maxsize\_w, maxsize\_h , 1)),

keras.layers.Dense(512, activation=tf.nn.relu),

keras.layers.Dropout(0.5),

keras.layers.Dense(128, activation=tf.nn.relu),

keras.layers.Dense(16, activation=tf.nn.relu),

keras.layers.Dense(2, activation=tf.nn.softmax)

])

bigger\_model2 = keras.models.Sequential([

keras.layers.Flatten(input\_shape = ( maxsize\_w, maxsize\_h , 1)),

keras.layers.Dense(1024, activation=tf.nn.relu),

keras.layers.Dropout(0.5),

keras.layers.Dense(512, activation=tf.nn.relu),

keras.layers.Dropout(0.5),

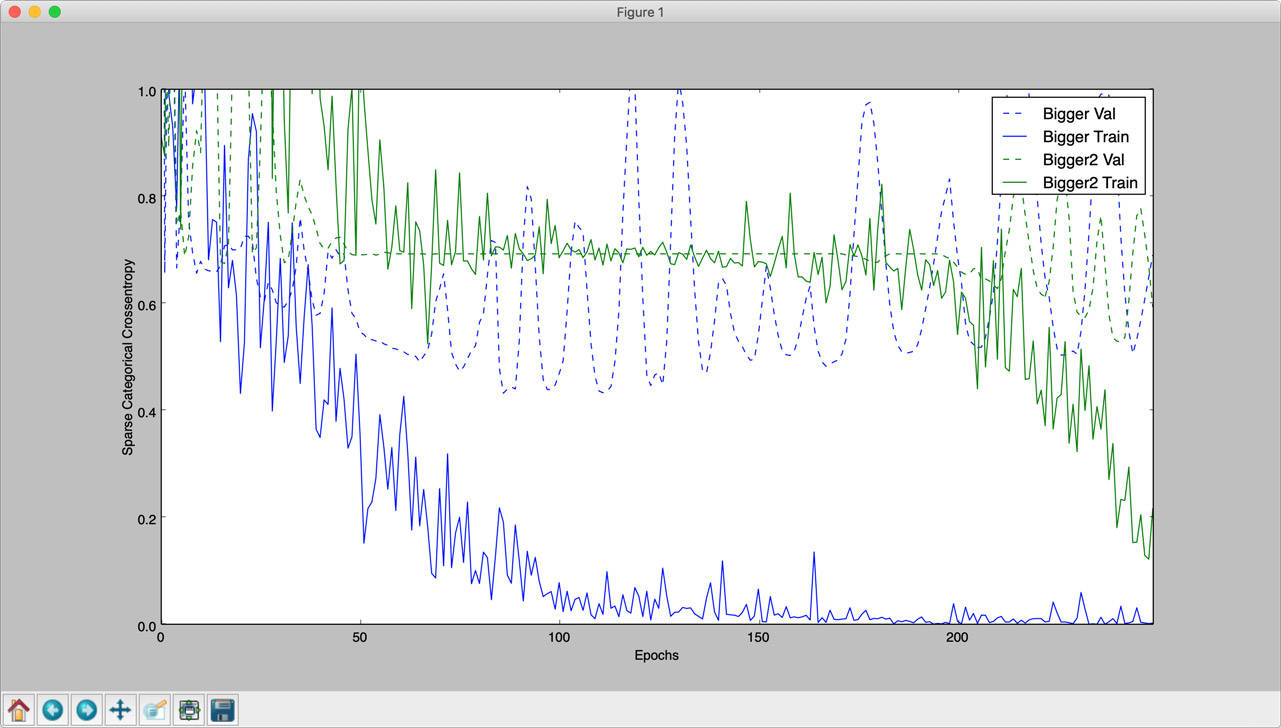
keras.layers.Dense(64, activation=tf.nn.relu),

keras.layers.Dense(16, activation=tf.nn.relu),

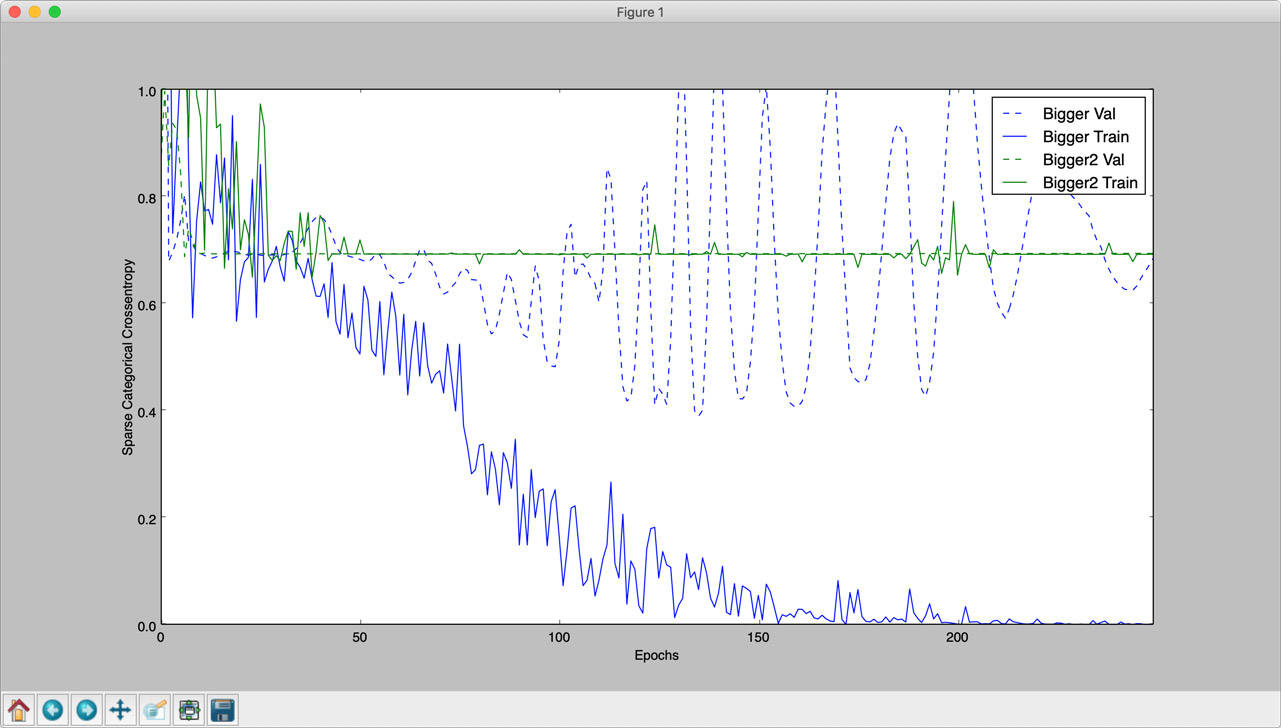
keras.layers.Dense(2, activation=tf.nn.softmax)

])

The following image shows the effect of applying dropout regularization.



During one run, the bigger model did not converge at all, even after 250 epochs. This is one of the side effects of applying dropout regularization.



**Lack of training data**

With only 26 or so training examples, we have done reasonably well. But for image processing, there are several techniques of data augmentation by applying some distortion to the original image and generating more data. For example, for every input image, we can have an invert color image added to our data set. To achieve this, the load\_image\_dataset function from Part 1 is modified as follows (it is also possible to add a randomly rotated image for each original image):

# invert\_image if true, also stores an invert color version of each image in the training set.

def load\_image\_dataset(path\_dir, maxsize, reshape\_size, invert\_image=False):

images = []

labels = []

os.chdir(path\_dir)

for file in glob.glob("\*.jpg"):

img = jpeg\_to\_8\_bit\_greyscale(file, maxsize)

inv\_image = 255 - img # Generate a invert color image of the original.

if re.match('chihuahua.\*', file):

images.append(img.reshape(reshape\_size))

labels.append(0)

if invert\_image:

labels.append(0)

images.append(inv\_image.reshape(reshape\_size))

elif re.match('muffin.\*', file):

images.append(img.reshape(reshape\_size))

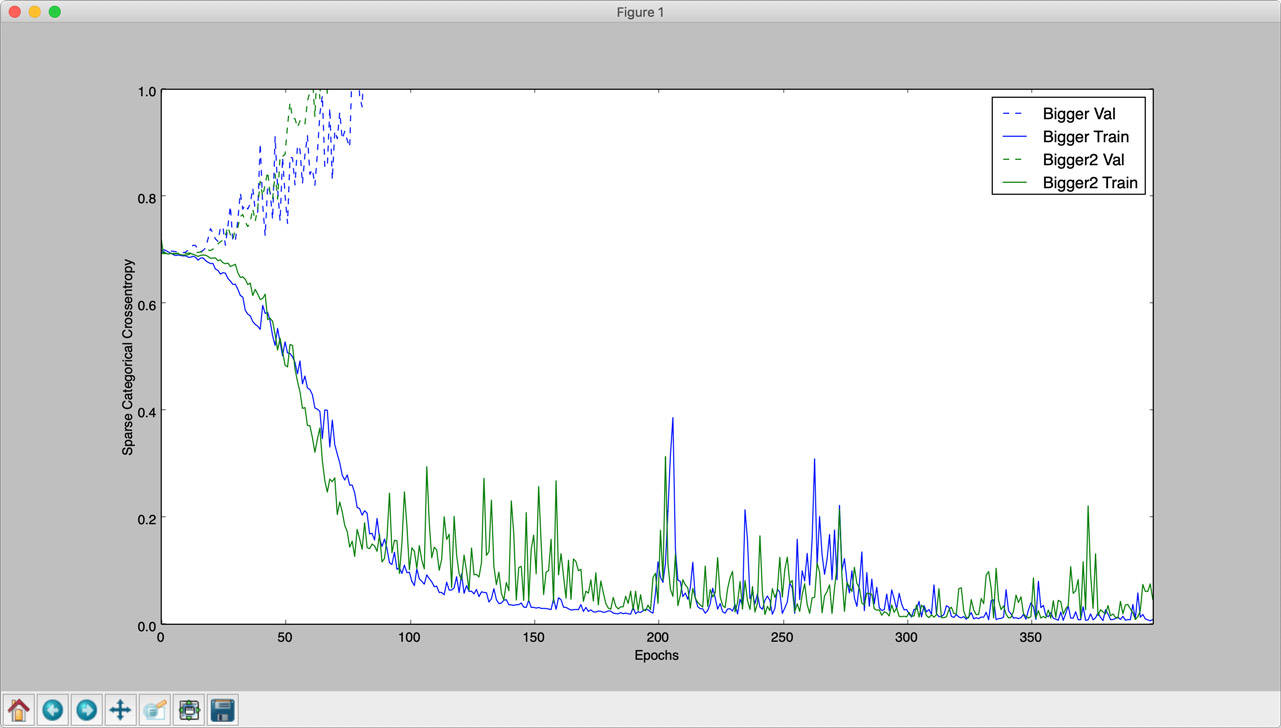
labels.append(1)

if invert\_image:

images.append(inv\_image.reshape(reshape\_size))

return (np.asarray(images), np.asarray(labels))

The effects of adding invert color images and randomly rotating images on training with dropout on is as follows. The size of the data set increased by 3X.



The result indicates that this has worsened the overfit of the data.

*Note: For data augmentation, Keras provides a built-in utility, keras.preprocessing.image.ImageDataGenerator, which will not be covered here.*

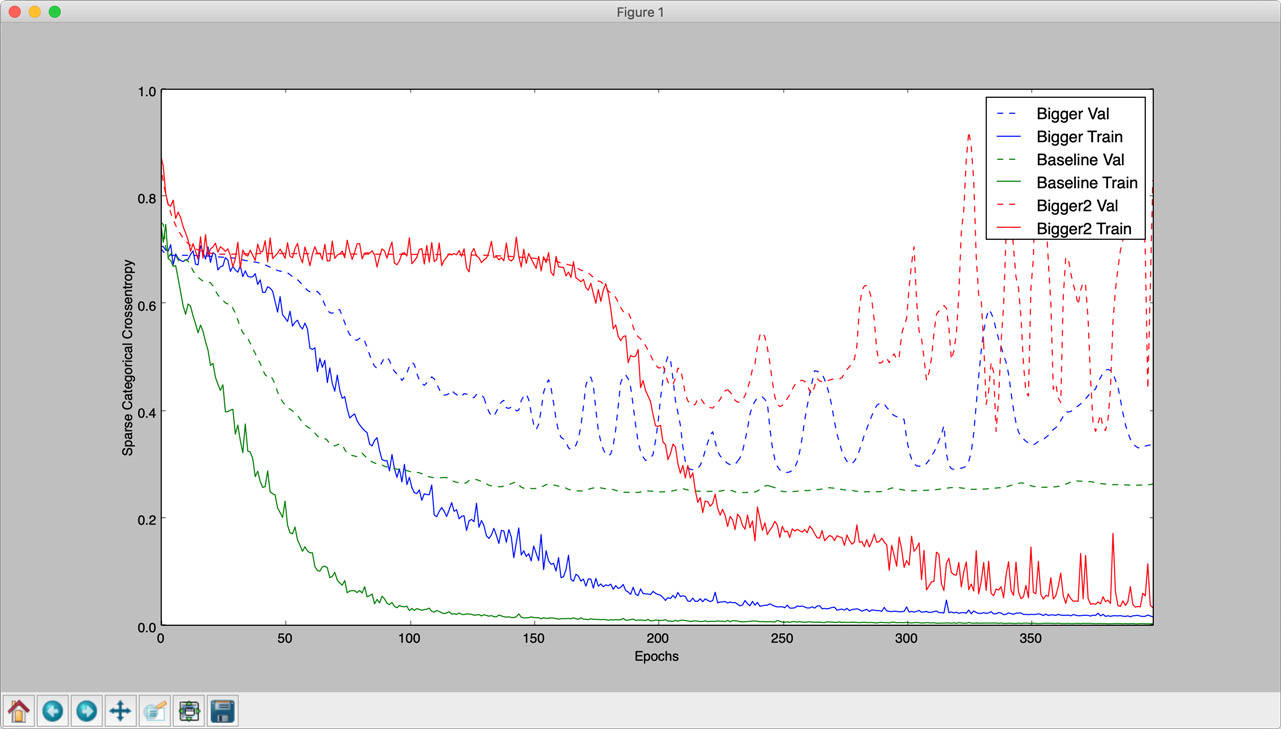
Another way to overcome the problem of minimal training data is to use a pretrained model and augment it with a new training example. This approach is called transfer learning. Since TensorFlow and Keras provide a good mechanism for saving and loading models, this can be quite easily achieved, but out of scope here.

**Conclusion**

On further testing with different models and activation functions, the best results were observed by using sigmoid as activation function and a dropout layer in our baseline model. Similar performance was observed with the relu activation function, but with sigmoid, the curve was smoother. And since the size of the image was reduced to 50×50, it improved the training time without impacting the performance of models.

Apart from the above, I have also tested a VGG-style multilayer CNN model and multiple variations of CNN models, but somehow the results were quite poor with it.

The following image shows the plot of the results from all three models.



Baseline model used:

baseline\_model = keras.models.Sequential([

keras.layers.Flatten(input\_shape = ( maxsize\_w, maxsize\_h , 1)),

keras.layers.Dense(128, activation=tf.nn.sigmoid),

keras.layers.Dropout(0.25),

keras.layers.Dense(16, activation=tf.nn.sigmoid),

keras.layers.Dense(2, activation=tf.nn.softmax)

])

baseline\_model.compile(optimizer=keras.optimizers.Adam(lr=0.001),

loss='sparse\_categorical\_crossentropy',

metrics=['accuracy','sparse\_categorical\_crossentropy'])

Output:

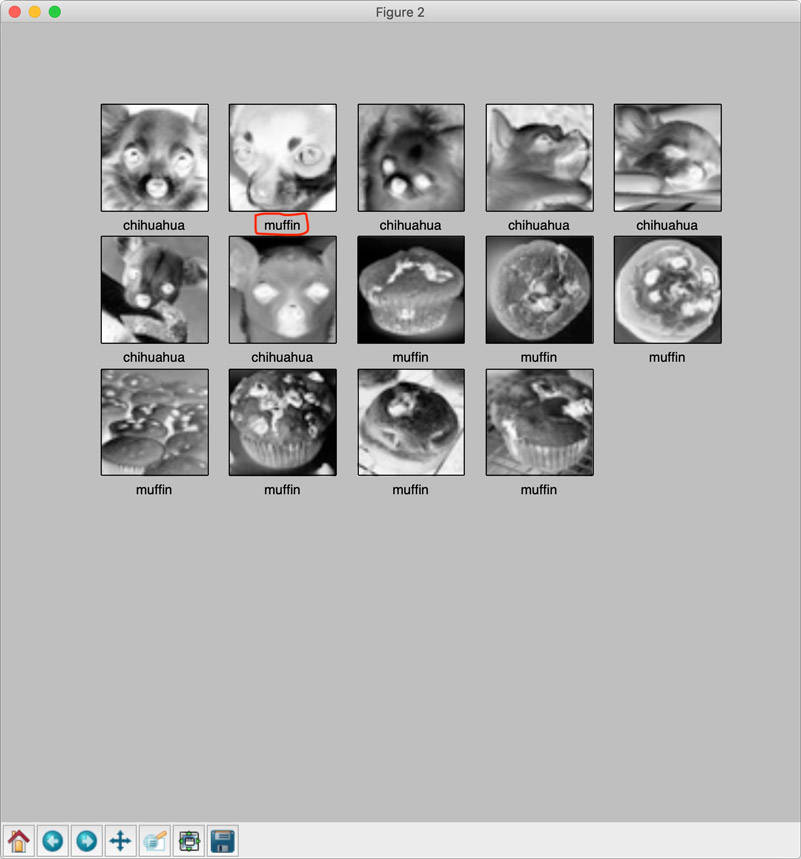
- 0s - loss: 0.0217 - acc: 1.0000 - sparse\_categorical\_crossentropy: 0.0217 - val\_loss: 0.2712 - val\_acc: 0.9286 - val\_sparse\_categorical\_crossentropy: 0.2712

Epoch 119/400

- 0s - loss: 0.0224 - acc: 1.0000 - sparse\_categorical\_crossentropy: 0.0224 - val\_loss: 0.2690 - val\_acc: 0.9286 - val\_sparse\_categorical\_crossentropy: 0.2690

Epoch 120/400

Results:



Next, I would like to improve my understandings of CNN and VGG-style networks for image recognition and even more advanced usages of neural networks.

Collections[#Open Source Data & AI Technologies](https://developer.ibm.com/collections/codait/)