Capstone Project

Machine Learning Engineer Nanodegree

Luis Cunha
Feb 15th 2019

I. DEFINITION

Project Overview

Flower classification, as well as of other living-organisms, is an important task used both for scientific purposes (taxonomy in research or applied sciences, such as agronomy) and leisure. It has traditionally been cumbersome, requiring specific and specialized knowledge, as well as careful and time consuming analysis of anatomical characteristics. Not long ago this knowledge was kept in books, either to be brought to the field, of to be used in the lab, this requiring field trips to collect specimens. More recently, it has been possible to more continently transport this knowledge to the field in a computer, or even on a smartphone. However, the process still remained the same as centuries ago, in which a person had to compare physical characteristics of the subject with the accumulated body of knowledge. This process can be greatly improved by automation, delegating the classification task to a computer. Image classification is a common and important application of Machine Learning (ML). It is used on a wide range of applications, from individuals identification for security proposes[1], to identifying road signs in the live video feed of self-driving cars[2]. I will develop a ML model to classify flowers from their images, while evaluating performance and cost (time and money) tradeoffs.

Problem Statement

A deep learning convolutional neural network model for flower classification needs to be built, which can be used as the basis of a flower classification system (such as an APP, or a REST API) by directly uploading an image and receiving the flower identification. The choices to be made to train such a model are numerous, from the datasets, to the training techniques, to the hardware choice. Deciding to train a model is just the first step. Many decisions must be made to balance the desired accuracy with the cost of training. With unlimited time and budget, one could collect a rich, high-quality, balanced dataset of flower images and fully train a neural network. However, that's seldom the case. Collecting images is laborious and expensive. Fortunately, several flower datasets of varying sizes and nature are available online. I've chosen to classify images typically found in the UK using a dataset provided by the Oxford University consisting of images of flowers from 102 distinct plant species. There exist other, larger datasets available, but the training becomes proportionally more costly on a cloud provider. I will explore the process of building a model to achieve the best accuracy vs training cost (time and money). Training this model involves:

- Defining an infrastructure strategy. There exist several services for one-click model training such as google Colab, fast.ai, FloydHub, etc. However, this project creates its own custom infrastructure on AWS, facilitated by terraform scripts.
- Defining a dataset (either collect the data or find an existing suitable dataset)
- Perform image pre-processing (such as organizing the images in class-named folders, normalize the pixel values, image augmentation such as rotations and translations, etc.). This project performing minimal image pre-processing prior to training. Most is accomplished at run-time using generators.
- Perform hyperparameter optimization
- Compare multiple CNN architecture and optimizer choices
- Fine tune the model.

Metrics

The evaluation metric to be used both with the benchmark and challenger models will be the categorical accuracy, which is appropriate for multiple class problem. This metric reflects the mean accuracy rate across all predictions (a single prediction being the class with the highest probability out of the 102 classes) and is calculated as

```
K.mean(K.equal(K.argmax(y_true, axis=-1), K.argmax(y_pred, axis=-1)))
```

This formula calculates the mean number of correctly predicted classes. The correctly predicted classes are determined as the instances where the class prediction and the true class label match. Since this is a categorical classification problem with multiple classes (more than 2), the predicted class is the one with the highest probability, as determined by `K.argmax(y, axis=-1)`.

II. Analysis

Data Exploration and Visualization

Datasets and Inputs:

There exist several datasets of images of flowers. The Kaggle 2018 FGCVx flower classification challenge dataset includes 1000 species and over 600000 images. Preliminary attempts to use this dataset revealed it to be cost prohibitive to train on a cloud service provider such as AWS (e.g. accuracy of a recent CNN architecture was only 70% after 2 days of training, for a cost of ~$50). For someone with their own GPU hardware , such as a consumer

NVidia 1080 or 2080 graphics card, this would be a fine dataset. However, Since this project runs on cloud computing platforms, it will use smaller, a budget friendlier dataset. The university of Oxford provides 2 smaller datasets, one of 17 species common in the UK, and a larger dataset of 102 species, with 40-258 images per category. This project will use the latter. The species labels are not provided by Oxford (only integer coded categories), but the true labels were deduced by GitHub user m-co and will be used (see the data sources in Appendix 1)

Figure 1a shows the distribution of images per class. There is some class imbalance in this dataset, but no effort will be made to equalize the number of images per class. As a future improvement one could use image augmentation to create the additional images required for each category to balance the classes. In this dataset, the number of images varies between 11 and 167, with most classes (interquartile range) having 29-66 examples (see descriptive statistics below). Figure 1b shows the distribution of the number of images to have a long tail of classes with a large number of images. Figures 2 and 3 show 20 examples of images from two distinct classes. As can be seen, the images within classes are very heterogeneous. Some show only flowers, others show just the leaves, while others show tree trunks. The image area occupied by plant parts also varies a lot. some show almost exclusively plant parts, while others show a lot of non-plant background. Some show a single plant, some others include multiple plants. This is expected the present a challenge to the classification algorithm and limit its maximum accuracy.

Distribution of images per class:

  Mean: 55.9
  Std: 30.7
  Min: 11
  Max: 167
  Percentiles: 10th: 34; 25th: 29; 50th:46; 75th:66, 90th:93

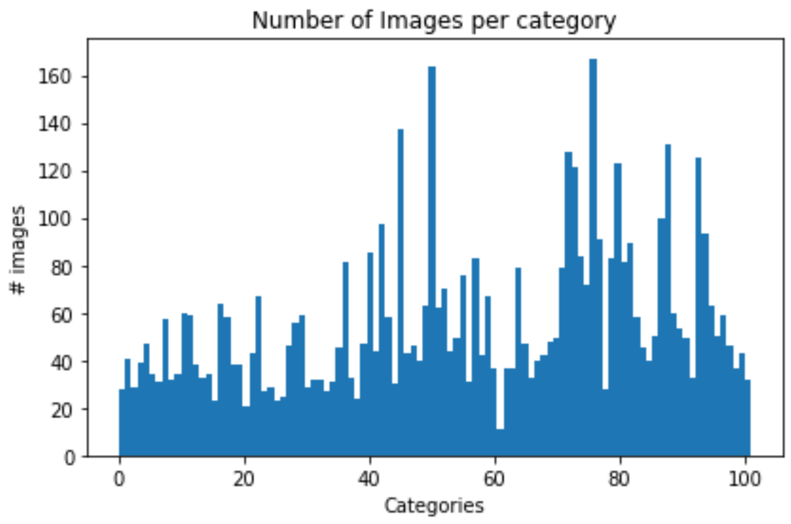


Fig 1a – Distribution of images per flower class. There are 102 classes and a little class imbalance is present. Most classes have between 30-60 images, while a few have as much as 160 examples. Only one class has less than 20 images.

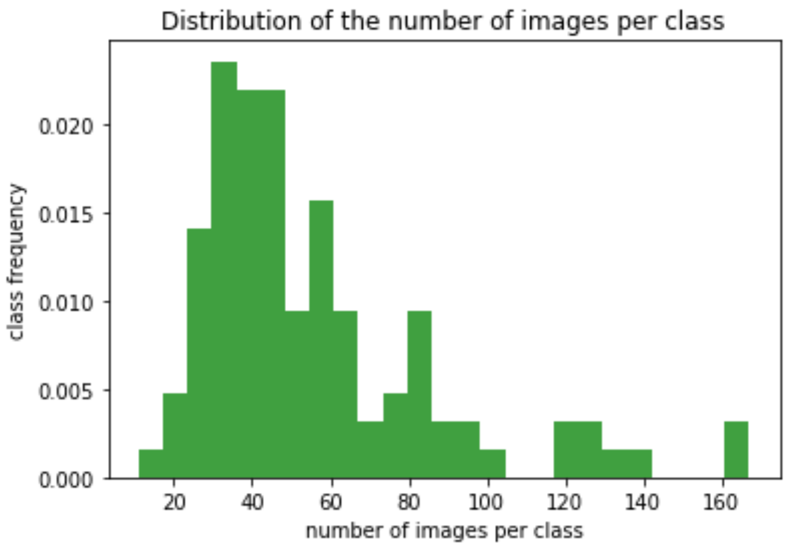


Fig 1b – Distribution of the number of images per flower class. Mean number of image is 55. A long tail of classes with a larger number of images is evident.

The following code displays a 5x4 grid of randomly selected images of a class, as shown in figures 2 and 3:

```
category = "/data/train/category_3/"
imgs = np.random.choice(os.listdir(category), size=20)

w = 10
h = 10
fig=plt.figure(figsize=(12, 12))
columns = 4
rows = 5
for i in range(1, columns*rows + 1):
    fig.add_subplot(rows, columns, i)
    img = mpimg.imread(category + imgs[i-1])
    imgplot = plt.imshow(img)
plt.show()
```

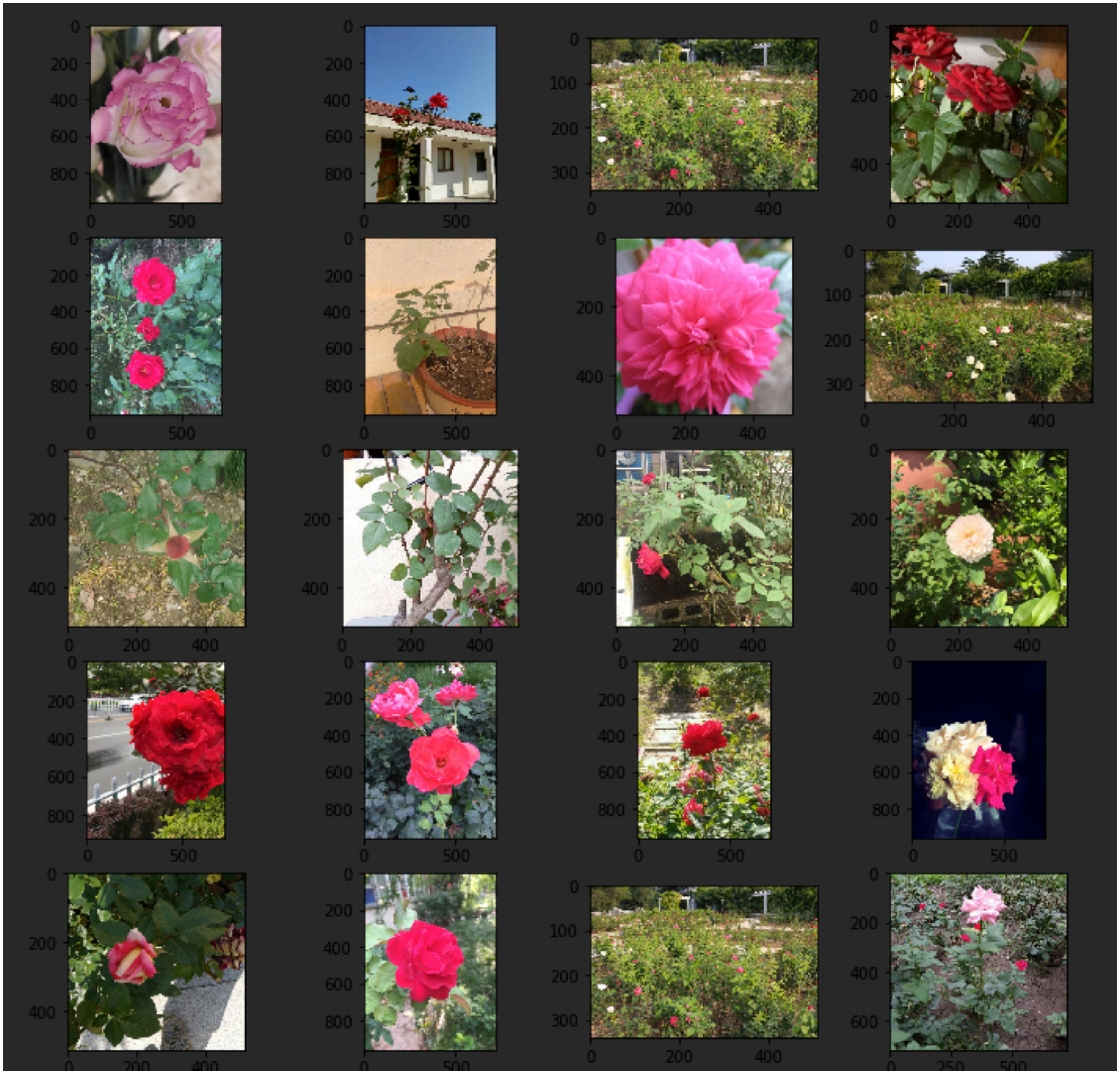


Fig. 2 – Examples of images from the Oxford102 dataset – class 1.

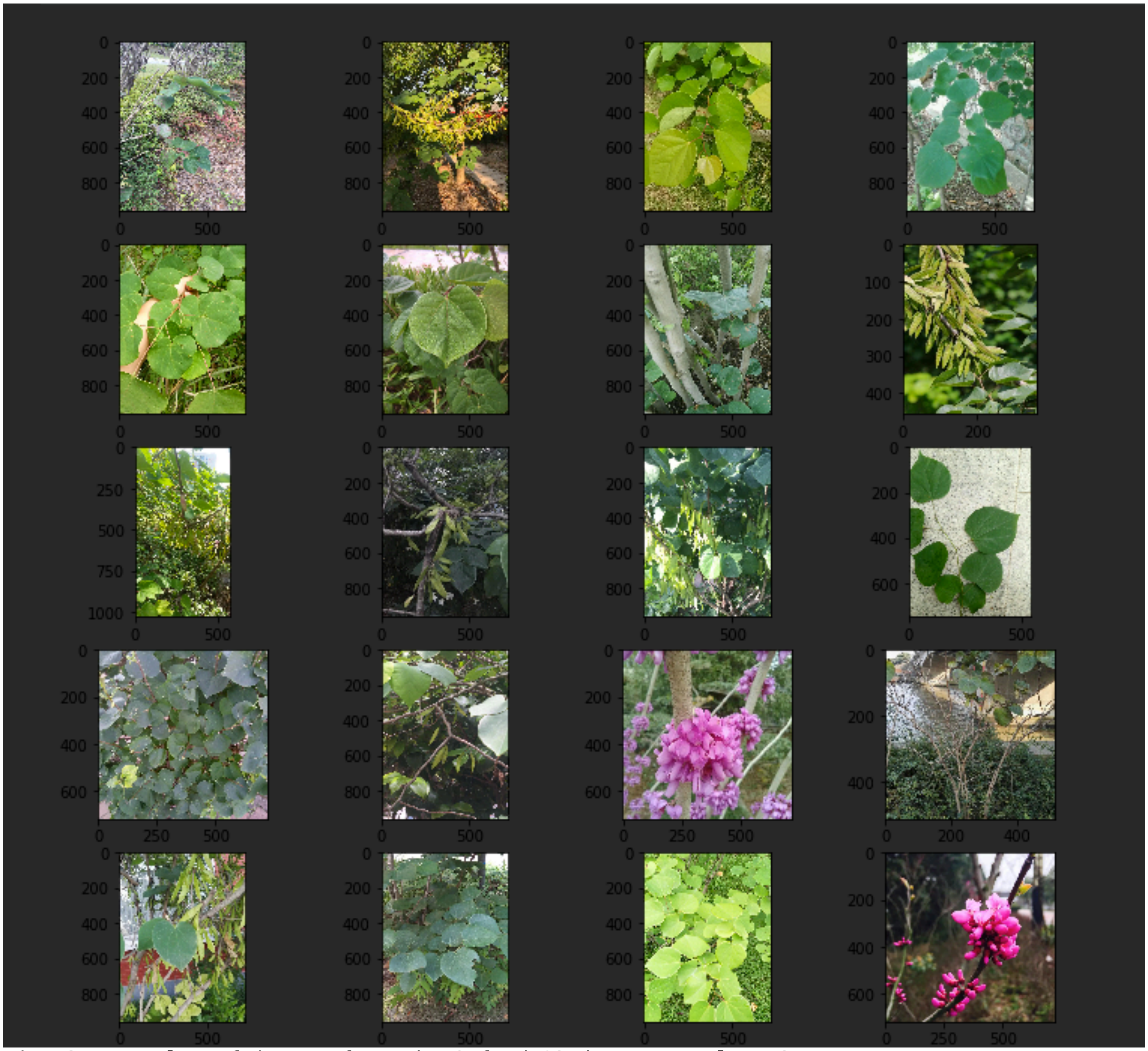


Fig. 3. Examples of images from the Oxford102 dataset – class 3.

Algorithms and Techniques

The algorithms of choice for image classification in the past decade have been convolutional neural networks (CNN). These outperform any other algorithm in image classification competitions, such as ImageNet. CNNs saw great development in the 1990s by Yann Lecun[3], but it was the easy access to

GPUs that allowed the recent gain in popularity of CNNs in computer vision problems. For the past decade, image classification competitors tried to outdo themselves each year, coming up with new, more complexed and creative CNN architectures, and each year the error rate has been steadily improving in double digit percentages[4].

Convolutional neural networks are regular neural networks, with layers, neurons, and learnable weights and biases. They also include forward and backpropagation, non-linear activation functions, a SoftMax function and the very end and a loss function. However, convnets make the assumption that the input is an image, which allows it to employ a few specific features that make it perform exceptionally well for computer vision. ConvNet architectures consist of convolutional, pooling, and full connected layers. Convolutional layers employ a convolutional function, that is, a function that is applied to another function. Specifically, a small filter is applied to the image pixels, significantly reducing the number of parameters. For a visual explanation, see https://goo.gl/7i3PEc . The rational for performing these convolutional is that neurons should be connected only to local regions, rather than far away regions of the image. Pooling layers between successive convolutional layers reduce the spatial size of the representation to reduce the number of parameters. For more information see this great review: http://cs231n.github.io/convolutional-networks/

The first attempt to solve this flower classification problem will involve a simple CNN as a benchmark model. I will then train a series of challenger models based on the ImageNet architectures (CNN architectures presented at the ImageNet competition) using **transfer learning.** Transfer Learning is a technique in which a model developed for one particular problem us used for another, more or less similar problem. This is possible because the different layers of a CNN recognize different levels of patterns granularity in the image. For instance, a model trained to recognize cats will consist of early layers that recognize edges, intermediate layers that might recognize more complex shapes, and deeper layers that might recognize cat features such as ears. Thus one could use the network almost as is to train a different dataset of cat images, possible of a different variety of cats, or retrain the top layers to recognize dog features. This patterns can reuse the early layers without modification since simple features like edges will be common to any type of image. Training of only the top layers will be much faster than training the whole network.

I'll perform hyperparameter optimization to try to achieved the maximal performance of each network architecture. Model parameters on the other hand are mostly fixed by the original architecture, except for the benchmark model, and do not change, except for the bottleneck layer. The hyperparameters include the choice of optimizer algorithm and optimizer-specific parameters (such as learning rate, decay and momentum), the number of epochs, the batch size, the step size, data augmentation with additional pre-processing beyond the pre-processing from the original architecture.

Benchmark

A simple neural network, not too dissimilar from the original LeNet[1], consisting of a 3 convolutional blocks, each consisting of a convolutional layer with ReLU activation and a maxpooling layer. The convolutional blocks are followed by a fully connected layer with dropout (to prevent overfitting), and a SoftMax layer the size of the number of classes to output class probabilities. The model was compiled with a RMSprop optimizer with default parameters.

III. Methodology

Data Preprocessing

The original dataset consists of images and a labels file. The labels are just numeric codes, not actual flower/plant species names. The plant names have been deduced by GitHub user m-co. The link can be found in the appendix 1.

Traditionally, in training a CNN model one would feed the algorithm pairs of image data and its label. However, Keras' image generator has a very convenient function (flow_from_directory) which allows the class labels to be provided as image directory names. Therefore, the first data pre-processing step was to segregate the images into their class label-named directories. Furthermore, the images were split into a train and a holdout test set of 20% of the dataset. An example of this directory structure might be:

```
> ~/training_data ❯
./iris:
    image_1.jpg
    image_2.jpg
    …
./red_rose:
    image_151.jpg
    image_152.jpg
    …
./…
```

All subsequent processing is done at run time by Keras, which include:

- Splitting the images training and validation sets
- The pixel values might be normalized (in the case of the benchmark model)
- In the case of models based on ImageNet architectures, the same pre-processing of the original network is applied
- Any processing for data-augmentation
- Images resizing: 224x224x3

Implementation

A) Model Implementation

1) Benchmark Model

The benchmark model is a simple convolutional network, somewhat inspired LeNet[1] consisting of 3 convolutional blocks followed by a fully connected layer, dropout to minimize overfitting, and a SoftMax layer to output class probabilities the same size as the number of data classes. Each block consists of a convolutional layer, uses the ReLU activation, and has a maxpooling layer. ReLU is the most common activation function in neural nets because it helps prevent vanishing gradients (gradients that become increasingly small) and it's sparse. The maxpooling layer provides down-sampling, picking the most important features. It prevents overfitting, and helps speed up computation. The model shown below (figure 4 and table 1) was compiled with a RMSprop optimizer (using optimizer default parameters).

The following code builds the benchmark model with Keras framework:

```
Model = Sequential()

#conv_layer1
model.add(Conv2D(32, (3, 3), input_shape=(256, 256, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

# conv_layer2
model.add(Conv2D(32, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

# conv_layer3
model.add(Conv2D(64, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())  # this converts our 3D feature maps to 1D feature vectors
model.add(Dense(256))
model.add(Activation('relu'))
model.add(Dropout(0.5))

model.add(Dense(n_categories, activation="softmax"))

model.compile(loss='categorical_crossentropy', optimizer=keras.optimizers.RMSprop,
              metrics=['accuracy'])
```

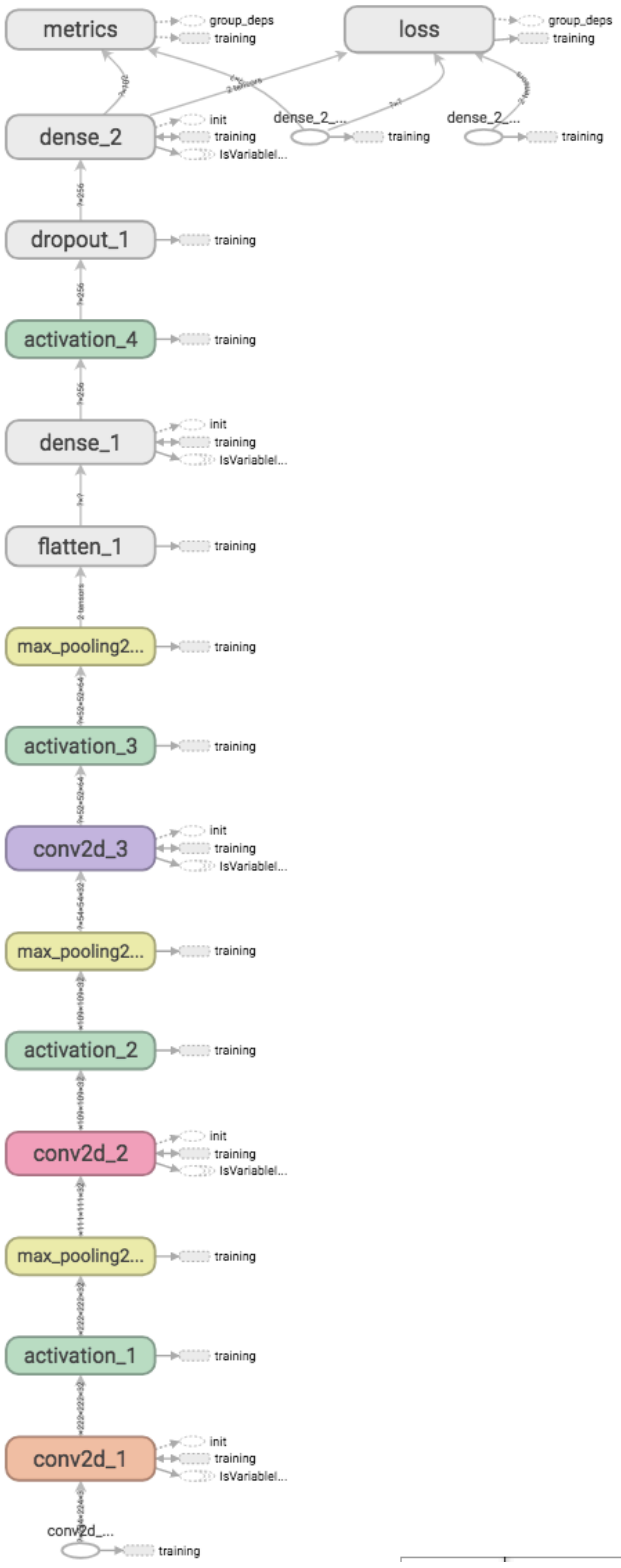


Fig. 4 – Schematic representation of the benchmark model (output from tensorboard)

```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 254, 254, 32)      896
activation_1 (Activation)    (None, 254, 254, 32)      0
max_pooling2d_1 (MaxPooling2 (None, 127, 127, 32)      0
conv2d_2 (Conv2D)            (None, 125, 125, 32)      9248
activation_2 (Activation)    (None, 125, 125, 32)      0
max_pooling2d_2 (MaxPooling2 (None, 62, 62, 32)        0
conv2d_3 (Conv2D)            (None, 60, 60, 64)        18496
activation_3 (Activation)    (None, 60, 60, 64)        0
max_pooling2d_3 (MaxPooling2 (None, 30, 30, 64)        0
flatten_1 (Flatten)          (None, 57600)             0
dense_1 (Dense)              (None, 256)               14745856
activation_4 (Activation)    (None, 256)               0
dropout_1 (Dropout)          (None, 256)               0
_________________________________________________________________
dense_2 (Dense)              (None, 102)               26214
=================================================================
Total params: 14,800,710
Trainable params: 14,800,710
Non-trainable params: 0
_________________________________________________________________
```

Table 1: Summary of benchmark model.

2) ImageNet Models

The challenger models are based on previously trained architecture using transfer learning. When using transfer learning there are four main cases:

1. The new dataset is small (a few thousand) and similar to original training data
2. The new dataset is small but different from original training
3. The new dataset is large (millions) and similar to original training data
4. The new dataset is large and different from original training data

The guidelines to apply transfer learning to these 4 scenarios suggest that the first only training of the bottleneck layer (the custom layers applied to the top of the ConvNet layers of the original model) . The second case, of a small dataset of different nature, should require discarding the top (higher level features) layer and train a bottleneck (fully connected) layer. The third case (large dataset/similar data) should require only replacing and training the bottleneck layer (fine tune). In the last case (large dataset, different data) should involve retraining the whole network.

For this project all the convolutional layers will be preserved and bottleneck layer will be replaced and retrained. Fine tuning will re-train the weights of the top convolutional layer after the training of bottleneck has been performed. This will use a very small learning rate since the model weights are already close to ideal.

The imagenet project (http://www.image-net.org/) is an image database resource for researchers and the public to develop the computer vision field. It runs an annual competition for object detection and image classification (http://www.image-net.org/challenges/LSVRC/). Several widely used CNN architectures have been introduced at his competition, starting with the revolutionary LeNet-5 in 1998 which outperformed all the other image classification methods at the time. I'll refer here to these architectures debuted at the ImageNet competition as ImageNet architectures, or ImageNet models interchangeably. Keras, through its applications module, provides implementation of these architectures, including their pre-trained weights. These networks have evolved overtime in their complexity, from the simple LeNet-5 with just 7 layers (of which 3 are convolutional). Since then architectures grew in complexity and creativity. AlexNet (2012) increased the number of layers and introduced stacked layers. Inception (2014) introduced a new element, the inception module, used batch normalization, and the RMSprop optimizer. While it increased the number of layers to 22, it actually reduced the number of parameters. VGG (2014) initially consisted of 16 layers, although VGG-19 uses 19 layers. ResNet (2015) introduced the concept of skip connections, allowing to train a network with 152 layers while keeping complexity low. For a more detailed summary of these networks, including diagrams, see this comprehensive blog post: https://medium.com/@sidereal/cnns-architectures-lenet-alexnet-vgg-googlenet-resnet-and-more-666091488df5. Keras' applications module supports 10 architectures and this project will compare their performance on the task of classifying the flower images in the oxford 102 dataset. Those architectures are: Xception, VGG16, VGG19, ResNet50, InceptionV3, InceptionResNetV2, MobileNet, DenseNet, NasNet, and MobileNetV2.

The code below shows how training is set up for ImageNet models. A convenience function loads the requested model and the associated image pre-processing function. Only the base layers are loaded (specified with the parameter include_top=False), as a flower classification specific bottleneck layer replaces the original fully connected layer. The input_size parameter allow the usage of any arbitrary image size for most of the networks instead of the original image sizes used to train the network. I choose to load the original weights, which will be frozen during the initial training. The bottleneck layers include kernel_regularizer and activity regularizer (two regularizers provided by Keras to combat overfitting), as well as batch normalization between the linear and non-linear final layer for the same purpose.

```
def get_model(network_name="inception_resnet_v2", image_size=(256, 256)):
    k.set_learning_phase(0)

    img_width, img_height = image_size

    if network_name == "vgg16":
        base_model = keras.applications.vgg16.VGG16(weights="imagenet",
                                                    include_top=False,
                                                    input_shape=(img_width, img_height, 3))
        input_processor = applications.vgg16.preprocess_input
    elif network_name == "vgg19":
        base_model = keras.applications.vgg19.VGG19(weights="imagenet",
                                                    include_top=False,
                                                    input_shape=(img_width, img_height, 3))
        input_processor = applications.vgg19.preprocess_input
    elif network_name == "inception_resnet_v2":
        base_model = keras.applications.inception_resnet_v2.InceptionResNetV2(weights="imagenet",
                                                                              include_top=False,
                                                                              input_shape=(img_width,
img_height, 3))
        input_processor = applications.inception_resnet_v2.preprocess_input
    elif network_name == "mobilenet_v2":
        base_model = keras.applications.mobilenet_v2.MobileNetV2(weights="imagenet",
                                                                 include_top=False,
                                                                 input_shape=(img_width, img_height,
3))
        input_processor = applications.mobilenet_v2.preprocess_input
    else:
        raise Exception("Make sure the network name is correct")

    for layer in base_model.layers[:]:
        layer.trainable = False

    # Add custom bottleneck Layer
    k.set_learning_phase(1)
    x = base_model.output
    x = Flatten()(x)
    x = Dense(1024, activation="relu", kernel_regularizer=regularizers.l2(0.01),
           activity_regularizer=regularizers.l1(0.001))(x)
    x = BatchNormalization()(x, training=True)
    predictions = Dense(nr_categories, activation="softmax")(x)

    model = Model(input=base_model.input, output=predictions)

    return model, input_processor
```

The model is then compiled, in this case with a Stochastic gradient Descent optimizer (with Keras' default parameters):

```
model, input_processor = get_model("vgg19")

model.compile(loss = "categorical_crossentropy",
              optimizer = keras.optimizers.SGD(),
              metrics=["accuracy"])
```

B) Run-time data preprocessing and image generators

The only data processing done prior to training was splitting the dataset into training/validation and test sets, and assigning of class names to image directories. All other pre-processing was done at runtime by Keras’ image generators. The following code demonstrates image runtime pre-processing. The first function, get_image_generator, returns an image generator with the desired validation fraction split (in this case 20% of the data will be used for validation) and optional data augmentation modifications. The preprocess_input function, specific to each ImageNet model, applies the same pre-processing done to the images used to train the original model.

```
def get_image_generator(image_augmentation=False, validation_split=0.2, preprocess_input=None):

    if not image_augmentation:
        return ImageDataGenerator(
                             #rescale=1. / 255,  # only if not applying preprocess_input function
                             preprocessing_function=preprocess_input,
                            validation_split=validation_split)

    return ImageDataGenerator(
        #rescale=1. / 255,  # only if not applying preprocess_input function
        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’,
        preprocessing_function=preprocess_input,
        validation_split=validation_split)
```

The training and validation datasets are constructed with the generator returned from the above function, as data is read from the class-named directories by the flow_from_directory function:

```
train_generator = train_val_datagen.flow_from_directory(
    data_dir,  # this is the target directory
    target_size=(img_width, img_height),  # image resizing
    batch_size=batch_size,
    class_mode=’categorical’,
    subset=”training”)

validation_generator = train_val_datagen.flow_from_directory(
    data_dir,  # target directory
    target_size=(img_width, img_height),  # image resizing
    batch_size=batch_size,
    class_mode=’categorical’,
    subset=”validation”)
```

The first positional parameter indicates the directory to read the data from. This directory contains the class-named image subdirectories. The subset parameter of flow_from_directory, used in conjunction with the validation_split parameter of the generator, allows to split the images into training and validation sets. The target_size specifies an image size for resizing, and the class_mode refers to the type of feature, which in this case is categorical. The batch size indicates how many images to load per training step.

These two generators are used by the model's fit_generator function to yield the pre-processed images to the model (yield here being the technical term referring to generators' version of a return statement):

```
model.fit_generator(
    train_generator,
    steps_per_epoch=num_train_img // batch_size,
    epochs=num_epochs,
    validation_data=validation_generator,
    validation_steps=num_val_img // batch_size)
```

Notice how both generators are passed in. The steps_per_epoch is the number of images to be seen until an epoch is considered complete. In this case this parameter assumes a value of the number of images divided by the batch size, thus in each epoch all images are used. Alternatively, one can use only a fraction of the images per epoch (e.g. half, or a quarter). In fact, this will be one of the hyperparameters to train.

C) Set up training

So far we've set up the model and the image generators which feed data to the model's fit function. We've seen the use of Keras' fit_generator in order to perform mini-batch training. That is, batches of images are used at a time, and the use of generators allows us to efficiently load the training data on demand (pre-loading the whole data in advance would not be possible if the data size was larger than the available memory). Training proceeds for as many epochs as requested in the initial setup. The number of epochs is actual a very critical hyperparameter. If training ends before the model stops learning we'll end up suboptimal model accuracy. If the number of epochs is too high, not only we'll be wasting computer resources (and money if training on a cloud service infrastructure), but we'll end up with a less than optimal model, as the model performance degrades if trained past the peak accuracy, as seen in figure 7. Ideally, we'd want to stop training as soon as the maximal performance is achieved. Keras' fit_generator has a callbacks parameter which accepts a list of callback functions. Keras provides several callbacks functions for common tasks, and we can also define our custom callbacks. One of those is the EarlyStopping callback function. Its job is to tell Keras to terminate training early, that is, before the requested number of epochs is reached. We’ll want to use this function to tell Keras to stop training once the model reached maximum accuracy. We won’t know the model accuracy has reached a peak until at least another epoch has passed. However, the accuracy might fluctuate up down while trending up. Therefore, we should wait a few epoch after the accuracy does not improve to decide if we’ve reached the maximum. The “patience” parameter of the EarlyStopping callback specifies how many epochs without improvement to wait before terminating (usually set a 3 to 5). The monitor parameter specifies which quantity to watch for (the validation accuracy in this case), the min_delta specifies the minimal difference in accuracy from subsequent epochs to consider it an improvement (we’re considering any improvement, as small as it might be, to be an improvement. Note that negative change is not considered an improvement). The mode can be set to min, max, or auto. In min mode the training will stop when the monitored quantity stops decreasing, and in max mode will stop when it stops increasing. In auto mode Keras decides to run in min or max mode depending on the name of the monitored quantity (e.g. if we monitor the accuracy, non-improvement means the quantity has stopped increasing, but if we monitor the loss, non-improvement is considered when this quantity has stopped decreasing). With the following callback we’ll wait for 5 epochs past the last improvement before we terminate training

```
Early_stopping = EarlyStopping(monitor=’val_acc’, min_delta=0, patience=5, verbose=1, mode=’auto’)
```

At this point our model has degraded since we trained it past its peak accuracy. We could take note of the number of epochs required to achieve the maximal accuracy and retrain for that many epochs. However, we don’t have to. Another convenient callback to use is the “Checkpoint”.

```
Model_checkpoint = ModelCheckpoint(“my_model.h5”, monitor=’val_acc’, verbose=1, save_best_only=True,
save_weights_only=False, mode=’auto’, period=1)
```

The ModelCheckpoint saves the current model in h5 format. Its parameters are self-explanatory. The “mode” has the same meaning as in EarlyStopping. “Period” determines how frequently to save the model (value of 1 means to save after each epoch). Importantly, the “save_best_only” instructs Keras to only save the best model. This is very convenient. Used together with EarlyStopping it allows us to train the model past the peak performance without degrading the saved model.

I will use two more callbacks: CSVLogger, which saves the training and validation accuracy and loss to a csv file, which can be used later for plotting, and Tensorboard, which writes log files to a directory monitored by the tensorboard server to plot the training progress in real time.

```
Csv_logger = CSVLogger(“my_model.csv”, append=True, separator=’;’)
```

```
tensorboard = TensorBoard(log_dir=”logs/my_model.log”, histogram_freq=0, write_graph=True,
write_images=True)
```

Setting up tensorboard monitoring is very simple. The tensorboard callback specifies the tensorboard log directory, which can be in the same machine running the training job, or a remote disk, such AWS’ EFS or EBS volumes (I use the latter because of the higher throughput speed compared to EFS. EFS is also more expensive, although it has the advantage of being able to be mounted on multiple machines simultaneously). The following section details how to set up and connect to a tensorboard server. The following code shows the fit_generator function used with all the callbacks included. The return value (history_callback) contains the accuracy values and can be plotted immediately after the training job ends.

```
Checkpoint = ModelCheckpoint(“my_model.h5”, monitor=’val_acc’, verbose=1, save_best_only=True,
save_weights_only=False, mode=’auto’, period=1)
early = Early Stopping(monitor=’val_acc’, min_delta=0, patience=3, verbose=1, mode=’auto’)
tensorboard = TensorBoard(log_dir=”logs/my_model.log”, histogram_freq=0, write_graph=True, write_images=True)
csv_logger = CSVLogger(”mymodel.csv”, append=True, separator=’;’)

history_callback = model.fit_generator(
            train_generator,
            steps_per_epoch=num_train_img // batch_size,
            epochs=n_epochs,
            validation_data=validation_generator,
            validation_steps=num_val_img // batch_size,
            callbacks = [checkpoint, early, tensorboard, csv_logger])
```

D) Infrastructure setup:

a. Terraform

Model Trainings are performed on AWS infrastructure on a P2 (GPU) instance. The infrastructure is treated as ephemeral, that is, each instance is terminated after each job, and a new one is started for a new job. Data is persisted on a permanent EBS volume that is attached/detached every time to each new instance. Following the principals of DevOps and infrastructure as code, terraform framework is used to facilitate provisioning the required infrastructure with one command. The companion GitHub repository to this report includes the terraform templates in the “deploy” directory. Once defined, provisioning and destroying the required resources on AWS is as simple as running the following commands, respectively:

```
> terraform apply
> terraform destroy
```

Updates to the infrastructure, such as updating the image AMI, are very simple to apply and require only modification of configuration files. For more information see the readme in the deploy directory.

b. Computer resources and virtual images (AWS Image AMI)

All training jobs are performed on AWS on p2.xlarge instances, which have 1 Tesla K80 GPU, 4 vPCUs, 61GB of memory, 2496 parallel processing cores and 12GB of GPU memory. At the time of this writing, it costs $0.90 per hour. The total cost of the AWS infrastructure for this project was about $1500.

On could certainly use a barebones image of any flavor of Linux and install all the packages necessary (Keras, python, jupyter notebook server, TensorFlow, NumPy, etc.). However there are pre-existing images geared towards data-science with all the common software packages and libraries pre-installed. This project used one such image: ami-00fc7481b84be120b (Deep Learning AMI (Ubuntu) Version 19.0).

c. Persistent Volume Storage

A large amount of disk space is required for the model training described here. Besides the size of the dataset, most storage was required by model checkpoint h5 files, each typically in the hundreds of MB, depending on the CNN architecture. This project treats infrastructure as ephemeral, that is, computing resources are discarded between jobs. Therefore, both the dataset and files generated by the training need to be persisted remotely. Since this projects runs on AWS' infrastructure, storage options such as EBS volumes, EFS, and S3 are available. S3 has a different storage model (object storage) that doesn't fit with the filesystem needs of the training algorithms, however it would be a valid option for backing up the data. EFS has the convenience of being possible to be mounted on multiple instances simultaneously. That would be of interested if we I performed parallel training jobs on multiple instances, but I'll run a single job at a time for this project. EFS is also the most expensive storage option of the three, and the network throughput is slower than EBS. This project used a 500GB gp2 (general purpose) EBS volume to full capacity, at a cost of $0.1/GB/month.

d. Jupyter notebook server

All training jobs were performed interactively on Jupyter notebooks.

The notebook server can be installed with pip or Conda (some AWS data science images come with it pre-installed). To quickly get started, cd to the data directory and start the server:

```
> jupyter notebook
```

The notebook can be accessed on localhost, port 8888. The server can accept configuration parameters in the command line, or a config file, such as to bind to different address/port, to specify a password, or to specify the working directory. I prefer to bind the notebook server on localhost and create an ssh tunnel for remote access, similarly to as described below for tensorboard.

e. tensorboard server

On the machine running the model training (or another machine with access to the tensorboard log directory if on EFS), start the tensorboard server:

```
tensorboard –logdir=path/to/log-directory
```

The dashboard is then accessible on the address localhost:6006. If running tensorboard on a remote machine, as is the case of all the training jobs described here (which run on AWS EC2), we can access it locally through a ssh tunnel:

```
ssh -i ~/.ssh/my_key.pem -N -L 6006:127.0.0.1:6006 user@remoteip
```

The –N flag indicates ssh to create the tunnel but not open a terminal into the remote machine. –L indicates the local port. 127.0.0.1 is address that tensorboard is bound to on the remote machine. user@remoteip are the username and remote machine IP address, and the –i flag accepts the ssh key to connect to the remote machine. Appendix 2 shows an example of several training jobs monitored by tensorboard.

E) Hyperparameter tuning

Hyperparameters are those parameters that are part of the training job, but are extrinsic to the model. Model parameters include the architecture choices, such as the number and size of layers, the filter and stride sizes, etc. Examples of hyperparameters include the batch and step sizes, the choice of optimizer and optimizer parameters. Since this project concerns itself with comparing the performance of multiple model architectures (10 ImageNet architectures + the benchmark model) the hyperparameter space is very large, making it computationally impractical. Instead of surveying the entirety of the hyperparameter space I will take a nested approach. At first I will optimize general hyperparameters (e.g. as batch and step size) that can be applied to all architectures, and freeze their values. This tuning is performed on a few, very distinct networks to avowing being trapped with results that are specific to one particular network. The values from these optimizations are then fixed as I move on to more specific tuning, e.g. of optimizer parameters. Fine tuning of hyperparameters with a lot of options, such as optimizer tuning, will also result in a very large hyperparameter space. Examples of strategies to search the parameter space include grid search, random search, Bayesian optimization, gradient based optimizations, etc. Grid search is an exhaustive search of all possible parameter combinations and can be very expensive computationally. Alternatively, one might search just a fraction of the search space. Here I'll perform random search by selecting 20% of the parameter combinations for each optimizer. For comparison purposes, the same optimizer parameters will be used across the different network architectures.

<u>Batch size</u> – Batch size refers to the number of training examples used in one iteration (epoch). One iteration is one pass through the neural network, including forward feed to make predictions and backpropagation to adjust the individual weights of each node, on each layer, based on the size of the errors made by the network. There are two batch modes: stochastic, in which one sample is used at a time, and minibatch, in which more than once example is used, but less than the size of the dataset. I will be using minibatch to train the CNNs. Choosing the batch size is still a matter of trial and error. Within a certain size range, smaller batch sizes will take longer to train, while larger sizers will train faster. However, due to vectorization, training 1 sample or training 10 samples might take about the same time. To determine the best batch size to this particular task, I trained two different network architectures: inception_resnet_v2 and VGG19, with batch sizes of 8, 16, 32, 64, 128, 256, and 512. Figure 5 shows the comparison of the training time of each network, the validation accuracy, and the loss, with respect to the batch size, using two optimizers: Stochastic Gradient Descent (SGD) and RMSprop, using their default parameters (more on optimizers ahead). There doesn't seem to be a clear pattern of dependency of these parameters on the batch size, except for with the inception_resnet / SGD combination. Batch sizes equal or larger to 256 exhausts the GPU's 12GB memory when training VGG19 with SGD. Based on these results, subsequent models will be trained with a batch size of 128.

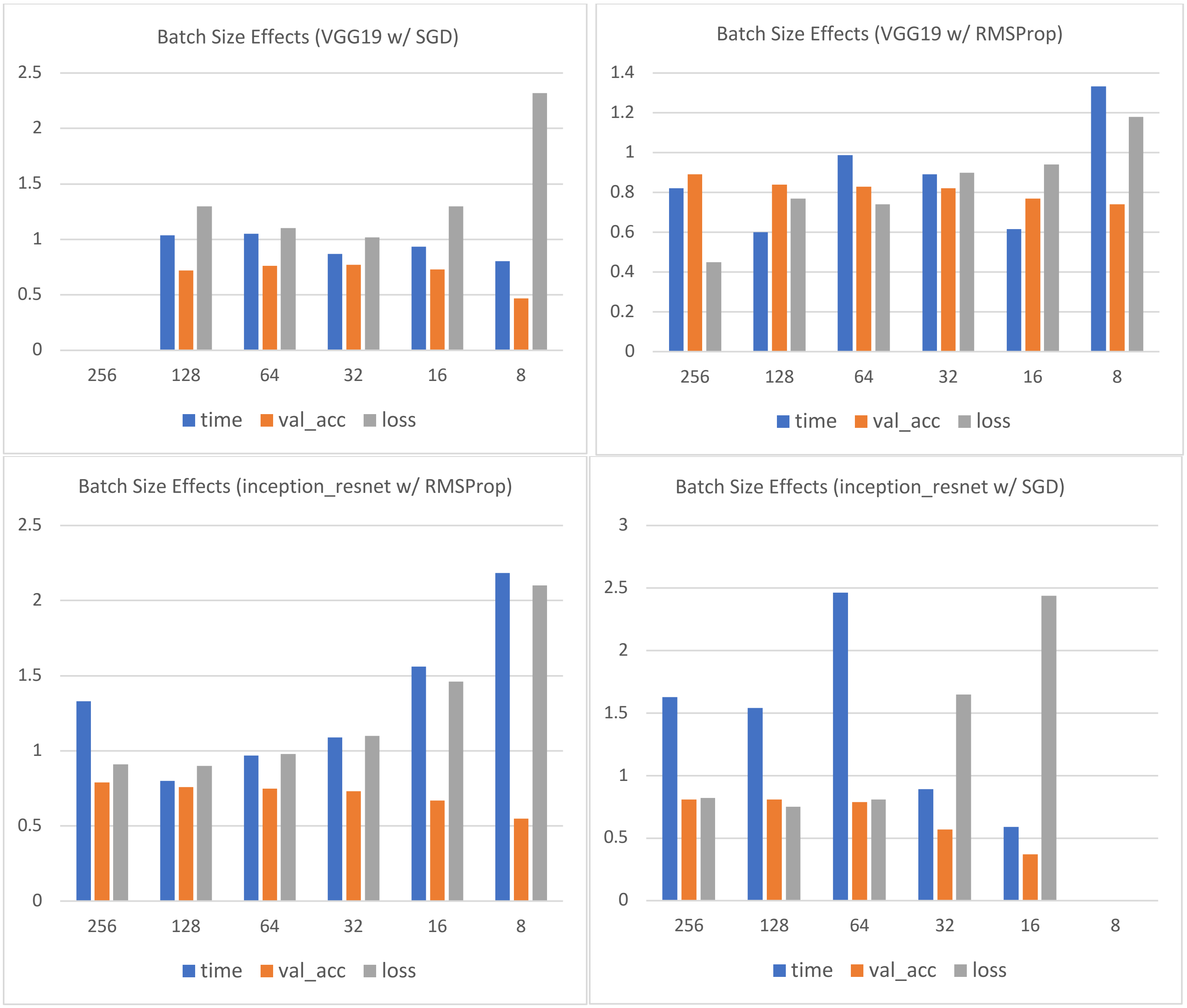


Fig. 5 – Batch Size optimization – Time, validation accuracy and validation loss as a function of batch size (time is value x 1000s). Top row: VGG architecture. Top Left – with SGD optimizer; Top right – with RMSprop optimizer; Lower row: Inception_Resnet_v2 architecture. Bottom left – with SGD optimizer. Bottom right – with RMSprop optimizer. Blue columns represent the time to reach epoch at which the network stops learning, as measured by no improvement in the validation accuracy. Orange column represents the maximum validation accuracy achieved, and in gray the lowest loss. The results indicate no clear trend on these 3 parameters as a function of the batch size, with perhaps an exception for the training time with the inception_resnet_v2 network and RMSprop as the optimizer. VGG19 with SGD has no data with a batch size of 256 because it exhausts the memory of the instance used for training.

Step Size – The step size controls how many samples are seen per epoch. In order for each epoch to “see” all data the step size must be set to the total amount of data divided by the batch size. The code below shows training with only half the data seen in each epoch (note that validation is performed on the entire dataset validation fraction), which is controlled by the fraction_of_data parameter (a value of 1 yields the full dataset and a value of 4 yields a quarter of the data, etc.). I compared training with the full dataset, half, and a quarter of the data seen per epoch. A inception_resnet_v2 architecture model was used the SGD optimizer with Keras’ default parameters, and batch size of 128. The performance of the model trained with half the data seen per epoch is similar to that the full dataset (0.76 vs 0.78, respectively). Training with only a quarter of the data resulted in lower accuracy (0.70). Halving the step size reduced the training time in half, from 80 to 45 seconds per epoch, thus also halving the training costs.

```
fraction_of_data = 2

history_callback = _model.fit_generator(
            train_generator,
            steps_per_epoch=num_train_img // batch_size // fraction_of_data,
            epochs=n_epochs,
            validation_data=validation_generator,
            validation_steps=num_val_img // batch_size,
            callbacks = [checkpoint, early, tensorboard, csv_logger])
```

Based on this results, subsequent training jobs will use a step size that results in the use of half the data per epoch in order to save time (and cost) without compromising performance.

Optimizer tuning – I’ll evaluate different optimizer choices and their parameters. Keras offers support for 7 optimizers. Briefly, these are: Stochastic Gradient Descent (SGD), RMSprop, Adagrad, Adadelta, Adam, Adamax, and Nadam. A description of these optimizers is beyond the scope of the project. A good explanation can be found here: http://ruder.io/optimizing-gradient-descent/. Keras recommends using the default parameters, except for the learning rate, which might be tuned freely. In order to assert the correctness of this suggestion, I will perform hyperparameter tuning of the parameters of the SGD and RMSprop optimizer in the context of two very distinct network architectures: VGG19 and mobilenet_v2.

A full grid search with multiple value choices for each optimizer’ parameters or the 4 optimizer/architecture combinations would be very computational intensive (and expensive). Instead, I’ll perform a random search, which randomly selects 20% of all the parameter combinations for tuning (the random seed will be the same among the different networks).

SGD supports leaning rate, decay, momentum, nesterov as tunable parameters, while RMSprop supports learning rate, decay, rho, and epsilon.

The learning rate is a parameter that controls the size of the weight updates in the gradient descent algorithm; decay controls a decrease in the size of the learning rate parameters as the training progresses; Momentum and Nesterov are parameters to accelerate the gradient descent algorithm in the relevant direction and dampens oscillations; Rho is the decay of the exponentially weighted average, which also has the effect accelerating the gradient descent algorithm in the right direction, and epsilon is a fuzz factor to prevent the weights from blowing up. The following code first defines the complete parameter grid for RMSprop. It then randomly selects 20% of the parameter combinations for tuning. A similar procedure was followed for SGD’s parameters.

```
import itertools
import numpy

seed = 42
np.random.seed(seed)

lr = [0.1, 0.01, 0.001]
rho = [0.5, 0.75, 0.9]
epsilon = [1e-7, 1e-6, 1e-4, 1e-2]
decay = [0.001, 0.01, 0.1, 0.2]

grid = list(itertools.product(lr, rho, epsilon, decay))
print(grid)
"""
>>> [(0.1, 0.5, 1e-07, 0.001),
     (0.1, 0.5, 1e-07, 0.01),
     (0.1, 0.5, 1e-07, 0.1),
     (0.1, 0.5, 1e-07, 0.2),
     (0.1, 0.5, 1e-06, 0.001),
     (0.1, 0.5, 1e-06, 0.01),
     (0.1, 0.5, 1e-06, 0.1),
     (0.1, 0.5, 1e-06, 0.2),
     (0.1, 0.5, 0.0001, 0.001),
     ...
     (0.001, 0.9, 0.0001, 0.2),
     (0.001, 0.9, 0.01, 0.001),
     (0.001, 0.9, 0.01, 0.01),
     (0.001, 0.9, 0.01, 0.1),
     (0.001, 0.9, 0.01, 0.2)]
"""

param_combinations = numpy.random.choice(range(len(grid)), int(len(grid)*0.2),
replace=False)
print(param_combinations)
"""
>>> array([ 11, 122,  37,  32,  70, 113,  77, 116, 134,  74, 143,  66,  41, 81, 102,
97,  62,  49,  51,  80,  36,  91, 65,  98,  76, 135, 40,  45])
"""
```

Image Augmentation – Image augmentation can be easily performed in Keras by specifying the transformations in the image generator. The following code applies rotation and zooming, which are evident in the images below: the top right column is very zoomed in, while the bottom right image has a 90 degree rotation (original images not shown)

```
image_datagen = ImageDataGenerator(
            rotation_range=90,
            width_shift_range=0.2,
            height_shift_range=0.2,
            shear_range=0.2,
            zoom_range=0.2,
            horizontal_flip=True,
            fill_mode='nearest',
            preprocessing_function=input_processor,
            validation_split=0.2)
```

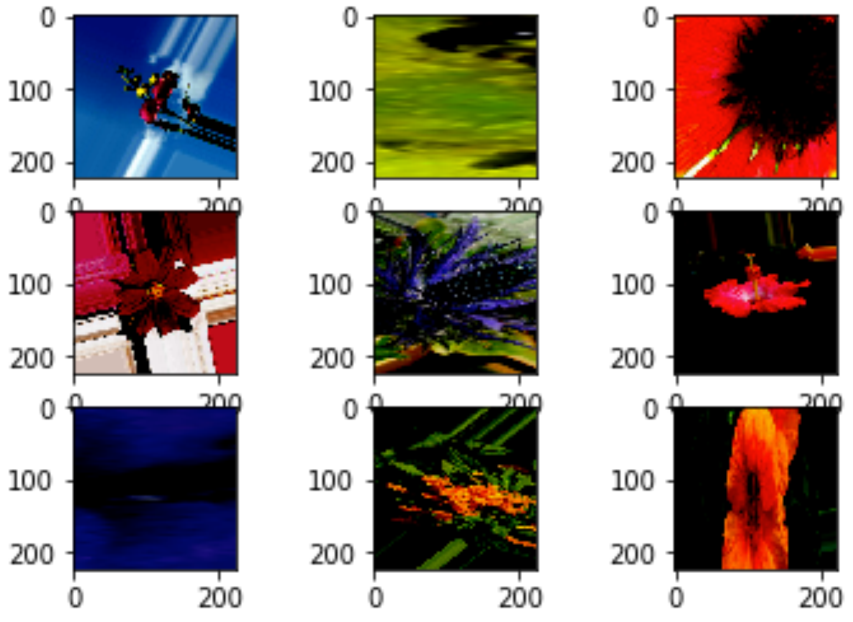


Fig. 6 – images with transformations applied by a image generator.

Figure 7 compares training of a mobilenet_v2 model with Adam optimizer with and without image augmentation (all other parameters being the same between the two training jobs). As can be seen, image augmentation does not improve the model metrics (validation accuracy/loss of 0.87/0.51 and 0.83/0.68 without and with image augmentation, respectively), and in fact seem detrimental. This result is consistent with studies that have shown traditional image augmentation techniques to be insufficient with current CNN architectures with a large number of parameters[5]. Therefore, image augmentation will not be used for subsequent training jobs.

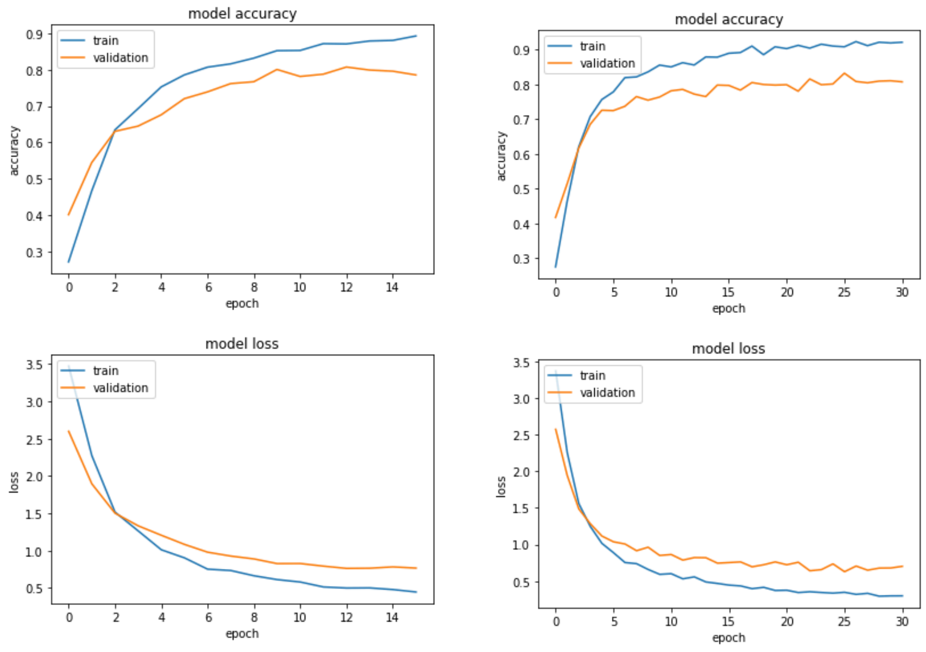


Fig. 7 – Classifier Training with (left) and without (right) image augmentation.

Refinement

Once the hyperparameter optimization has been concluded and the best model selected, a refinement will be performed by "thawing" the previously frozen layers of the top convolutional block (above layer 480 in the case of the model shown below. Fine tuning will use the stochastic gradient descent optimizer with a very small learning rate (since the model is already very optimized in the previous steps).

First load the previously trained model and weights:

```
from keras.models import load_model
model = load_model('path_to_model_h5_file')
```

Then set the last convolutional layers (and bottleneck layer) to trainable:

```
for i, layer in enumerate(model.layers):
    if i > 480:
        layer.trainable = True
    else:
        layer.trainable = False
```

Finally, compile the model:

```
model.compile(optimizer=optimizers.SGD(lr=0.0001, momentum=0.9), loss='categorical_crossentropy',
metrics=['accuracy'])
```

and train as previously:

```
history = train_model(params, model, (train_generator, validation_generator))
```

IV. Results

Model Evaluation and Validation

Benchmark model: The Benchmark model achieved a maximal accuracy of 0.53/0.44 without and with image augmentation, respectively

All ImageNet Architectures / optimizer combinations (with optimizer default parameters): Model training was performed with the previously optimized step size and batch size. No image augmentation as also previously determined. Table 2 shows a matrix of the performance of all ImageNet network architectures compiled with each optimizer supported by Keras, with default options. Densenet201 stands out as the best performing architecture, and it's best paired with the Adam optimizer, which achieves not only the highest validation accuracy, but also the lowest validation loss. MobileNet (version 1 and 2) performs slightly worse than Densenet201, but it trains twice to three times faster. Nasnet not only performed the worse, it also took the longest to train (with epoch times in the order of 90-100s versus the average of about 30s. MobileNet (v1 and 2) took about 9-15s per epoch).

When it comes to the optimizers, SGD and Adadelta seem to perform the worse among all the optimizers. Figure 8 Shows an example of accuracy and loss plots on the train and validation datasets, with the MobileNet architecture and RMSprop optimizer, which is typical of most training instances. As can be seen, the initial learning curve is fairly steep and accuracy reaches a maximum in just a few epochs. The model seems a bit overfitted, as the train accuracy is higher than the validation accuracy. However, techniques to reduce overfitting (increasing dropout, use more data through image augmentation) were not successful (results now shown).

Two Optimizer Hyperparameter optimization with two architectures: In addition to training these networks with each optimizer's default parameters, I performed a random search of the SGD and RMSprop optimizer's hyperparameters in the context of 2 architectures (VGG19, and ImageNet_V2). Note that Keras recommends using the default parameters, except the learning rate, which might be freely trained. The results in tables 3 and 4 suggest that this assertion is correct. The best parameter combinations were those close to the default values. Other combinations performed similarly or worse, but none better.

Fine Tuning: Fine tuning of the top performing model (densenet201 architecture with trained with Adam optimizer) in which the top convolutional layer’s weights were set to trainable, increased the accuracy to 93% while the model loss decreased to 0.27 (figure 9).

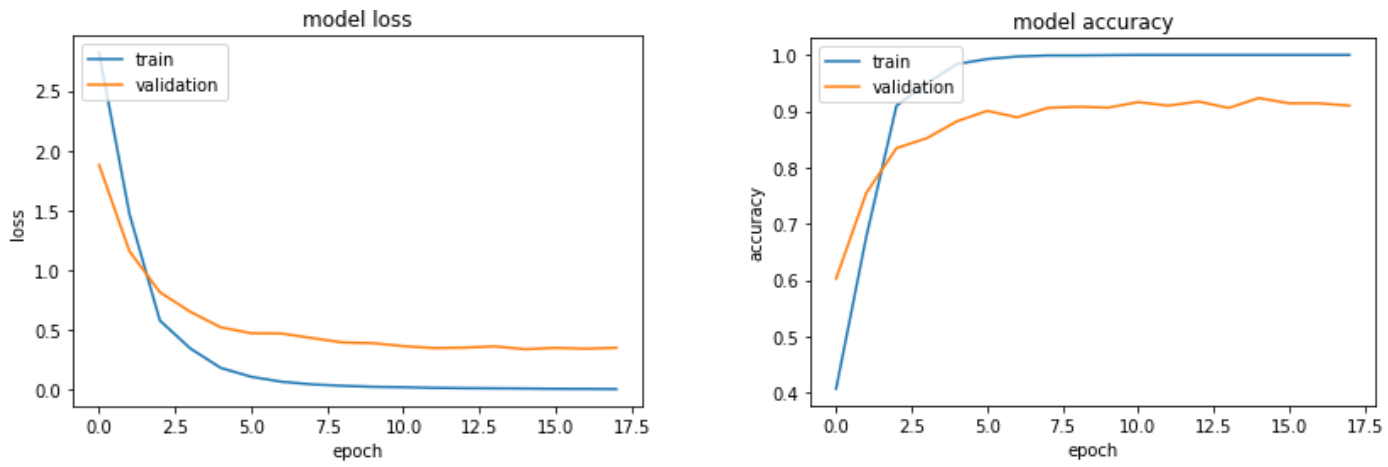


Fig. 8 – Plot of Model validation loss (above) and accuracy (below) on train and validation datasets (densenet201 architecture + Adam optimizer). Maximum accuracy achieved was 0.923, with a loss of 0.35 after training for 15 epochs. Training time was about 525s (35s per epoch)

	mobilenet_v2	vgg16	vgg19	inception_resnet_v2	mobilenet	xception	resnet50	inception_v3	densenet201	nasnet
SGD lr=0.001, momentum=0.0, decay=0.0, nesterov=False	0.81	0.64	0.7	0.74	0.8	0.8	0.8	0.73	0.87	0.76
RMSProp lr=0.001, rho=0.9, epsilon=None, decay=0.0	0.88	0.83	0.82	0.77	0.89	0.79	0.88	0.8	0.91 (0.40)	0.75
Adagrad lr=0.01, epsilon=None, decay=0.0	0.86	0.82	0.825	0.79	0.78	0.78	0.86	0.82	0.9 (0.52)	0.76
Adadelta lr=0.01, rho=0.95, epsilon=None, decay=0.0	0.73	0.59	0.58	0.62	0.66	0.73	0.75	0.61	0.82	0.69
Adam lr=0.001, beta_1=0.9, beta_2=0.999, epsilon=None, decay=0.0, amsgrad=False	0.88	0.85	0.83	0.74	0.91 (0.43)	0.82	0.87 (0.53)	0.81	0.923 (0.35)	0.75
Adamax lr=0.001, rho=0.9, epsilon=None, decay=0.0	0.86	0.82	0.81	0.79	0.89	0.82	0.87	0.8	0.90 (0.44)	0.73
Nadam lr=0.001, rho=0.9, epsilon=None, decay=0.0	0.87	0.84	0.84	0.77	0.89	0.82	0.89	0.82	0.91 (0.38)	0.74

Table 2 – validation accuracy of all 10 convnets with all 7 optimizers supported by Keras (with default parameters). Values in red are the top scoring combinations with validation accuracy above 90% (the value in parentheses indicates validation loss).

#	1	11	15	22	25	28	30	32	36	40	44	60	66	68
params (lr, decay, momentum, nesterov)	(0.1, 0.5, True, 0.01),	(0.1, 0.75, True, 0.2),	(0.1, 0.75, False, 0.2),	(0.1, 0.9, False, 0.1),	(0.01, 0.5, True, 0.01),	(0.01, 0.5, False, 0.001),	(0.01, 0.5, False, 0.1),	(0.01, 0.75, True, 0.001),	(0.01, 0.75, False, 0.001),	(0.01, 0.9, True, 0.001),	(0.01, 0.9, False, 0.001),	(0.001, 0.75, False, 0.001),	(0.001, 0.9, True, 0.1),	(0.001, 0.9, False, 0.001),
vgg19	0.76	0.56	0.55	0.59	0.72	0.74	0.74	0.77	0.77	0.81	0.8	0.64	0.36	0.69
resnet_inception_	0.31	0.62	0.75	0.62	0.73	0.76	0.76	0.79	0.77	0.8	0.8	0.66	0.78	0.75
mobilenet_v2	0.86	0.77	0.73	0.77	0.78	0.82	0.82	0.83	0.83	0.86	0.87	0.72	0.62	0.78

Table 3 – SGD hyperparameter optimization by random search experiments. It validates that Keras' statement that the default parameters are optimal.

#	11	32	36	37	40	10-Feb	44	45	49	51	60	62	65	66	70
params(lr, rho, epsilon, decay)	**(0.1, 0.5, 0.0001, 0.2)**	**(0.1, 0.9, None, 0.001)**		**(0.1, 0.9, 1e-06, 0.01)**	**(0.1, 0.9, 0.0001, 0.001)**	**(0.1, 0.9, 0.0001, 0.01)**		**(0.1, 0.9, 0.01, 0.01)**	**(0.01, 0.5, None, 0.01)**	**(0.01, 0.5, None, 0.2)**	**(0.001, 0.75, False, 0.001)**	**(0.01, 0.5, 0.01, 0.1)**	**(0.01, 0.75, None, 0.01)**	**(0.01, 0.75, None, 0.1)**	**(0.01, 0.75, 1e-6, 0.1)**
mobilenet	0.74	0.3	0.3	0.57	0.82919	0.60455	0.86749	0.78	0.82195	0.86232	0.71532	0.81677	0.81	0.86232	0.84679
vgg19	0.75155	0.29607	0.35818	0.50932	0.2627	0.48965	0.70911	0.76087	0.77329	0.81738	0.82715	0.77433	0.78675	0.83644	0.87474
#	74	76	77	80	81	91	97	98	102	113	116	122	134	135	143
params(lr, rho, epsilon, decay)	**(0.01, 0.75, 0.0001, 0.1)**	**(0.01, 0.75, 0.01, 0.001)**	**(0.01, 0.75, 0.01, 0.01)**	**(0.01, 0.9, None, 0.001)**	**(0.01, 0.9, None, 0.01)**	**(0.01, 0.9, 0.0001, 0.2)**	**(0.001, 0.5, None, 0.01)**	**(0.001, 0.5, None, 0.1)**	**(0.001, 0.5, 1e-06, 0.1)**	**(0.001, 0.75, None, 0.01)**	**(0.001, 0.75, 1e-06, 0.001)**	**(0.001, 0.75, 0.0001, 0.1)**	**(0.001, 0.9, 1e-06, 0.1)**	**(0.001, 0.9, 0.01, 0.2)**	**(0.001, 0.9, 0.01, 0.2)**
mobilenet	0.84369	0.86128	0.83954	0.78986	0.83954	0.78	0.89027	0.88	0.85921	0.89234	0.87474	0.86	0.853	0.84	0.74741
vgg19	0.82712	0.85404	0.85507	0.76708	0.80331	0.77832	0.81781	0.77743	0.78516	0.82919	0.82816	0.7795	0.78261	0.74948	0.5766

Table 4 – RMSprop hyperparameter optimization by random search experiments. It validates that Keras' statement that the default parameters are optimal.

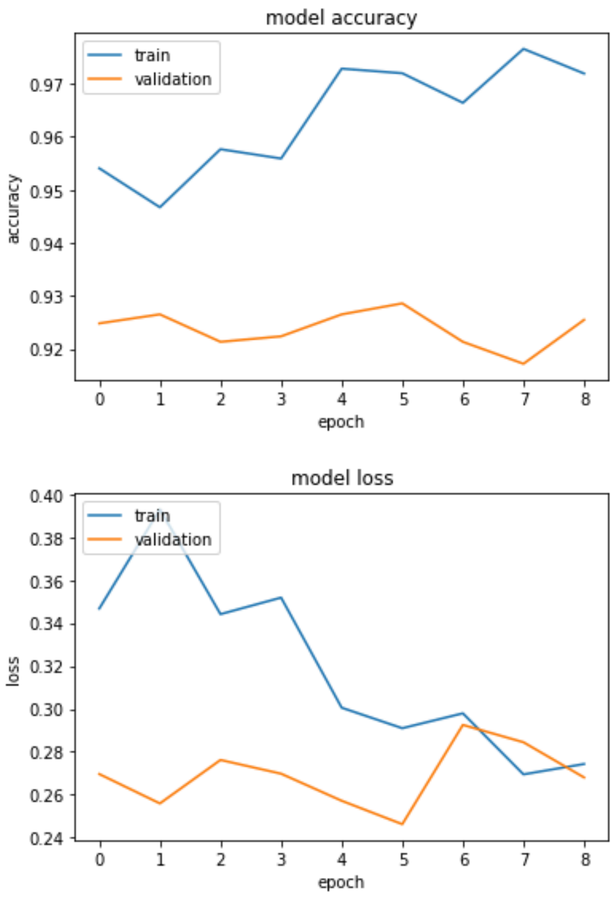


Fig. 9 – Model accuracy (above) and loss (below) of fine tuning training of a densenet201 model with SGD optimizer (lr=0.0001).

The code below calculates the model accuracy in the test (holdout) set, which has not been seen by the model up to this point. The fine-tuned model has a test accuracy of 0.79. It is significantly lower than the 0.93 validation accuracy, indicating that the model is overfited.

```
from keras.models import load_model

# use the same seed as used to train the model
seed = 7

n_test_images = 2441
batch_size = 128

# load the previously trained model
model =
load_model("/data/oxford102/experiments/1549845691.8560944/densenet201_1549845691.8560944.h5")

# apply the same image processing as in training
input_processor = applications.densenet.preprocess_input
train_val_datagen = ImageDataGenerator(preprocessing_function=input_processor)

test_data_dir = "/data/oxford102/test/"

# get a image generator object
predict_generator = train_val_datagen.flow_from_directory(
            test_data_dir,
            target_size=(img_width, img_height),
            batch_size=128,
            class_mode=None,
            color_mode="rgb",
            shuffle=False,
            seed=seed)

predict_generator.reset()
probabilities = model.predict_generator(predict_generator, n_test_images / batch_size, verbose=1)
>>> 20/19 [================================] - 38s 2s/step

# probabilities is an 2-dimentional array of n images, with probabilities for each of the 102 classes
probabilities.shape
"""
>>> (2441, 102)
"""

probabilities
"""
>>> array([[9.66245711e-01, 1.33186887e-07, 7.35159972e-07, ...,
        7.31853015e-06, 2.62090907e-05, 9.98339829e-07],
       [9.99977946e-01, 1.34325060e-11, 1.85547588e-09, ...,
        4.67980676e-09, 2.08144968e-09, 2.58107025e-09],
"""


# get the class prediction as the class with maximal probability
predicted_class_indices=np.argmax(probabilities,axis=1)

predicted_class_indices
"""
array([ 0,  0,  0, ..., 81, 87, 31])
"""
```

```
# get the image generator labels
labels = (predict_generator.class_indices)
_labels = dict((v,k) for k,v in labels.items())

_labels
"""
{0: 'alpine_sea_holly',
 1: 'anthurium',
 2: 'artichoke',
 3: 'azalea',
 4: 'ball_moss',
 5: 'balloon_flower',
 6: 'barbeton_daisy',
 ...
 97: 'water_lily',
 98: 'watercress',
 99: 'wild_pansy',
 100: 'windflower',
 101: 'yellow_iris'}
"""

# get the predictions in a list, sequentially as yielded by the image generator
predictions = [_labels[k] for k in predicted_class_indices]

predictions
"""
['alpine_sea_holly',
 'alpine_sea_holly',
 'alpine_sea_holly',
 'alpine_sea_holly',
 'alpine_sea_holly',
 'alpine_sea_holly',
 'alpine_sea_holly',
 'alpine_sea_holly',
 'globe_thistle',
 'alpine_sea_holly',
 'alpine_sea_holly',
 'alpine_sea_holly',
 'anthurium',
 'anthurium',
  ...]

"""

import pandas as pd

# save the predictions for each filename
filenames = predict_generator.filenames

results=pd.DataFrame({"Filename":filenames,
                      "Predictions":predictions})

results.to_csv("predictions_1549339395.8436813.csv",index=False)
```

```
results
""""
            Filename                          Predictions
0  alpine_sea_holly/image_06969.jpg   alpine_sea_holly
1  alpine_sea_holly/image_06970.jpg   alpine_sea_holly
2  alpine_sea_holly/image_06983.jpg   alpine_sea_holly
3  alpine_sea_holly/image_06993.jpg   alpine_sea_holly
4  alpine_sea_holly/image_06994.jpg   alpine_sea_holly
5  alpine_sea_holly/image_06996.jpg   alpine_sea_holly
6  alpine_sea_holly/image_06998.jpg   alpine_sea_holly
7  alpine_sea_holly/image_07002.jpg   alpine_sea_holly
8  alpine_sea_holly/image_07004.jpg    globe_thistle
...
2438   yellow_iris/image_06389.jpg    silverbush
2439   yellow_iris/image_06392.jpg    sweet_pea
2440   yellow_iris/image_06394.jpg    daffodil

""""

# the filesname list has the same order as the predicitons list
filenames
""""
['alpine_sea_holly/image_06969.jpg',
 'alpine_sea_holly/image_06970.jpg',
 'alpine_sea_holly/image_06983.jpg',
 'alpine_sea_holly/image_06993.jpg',
 'alpine_sea_holly/image_06994.jpg',
 ...
 ]
""""

# get the class name from the filenames

_filenames = list(map(lambda x: x.split("/")[0], filenames))

_filenames
""""
['alpine_sea_holly',
 'alpine_sea_holly',
 'alpine_sea_holly',
 'alpine_sea_holly',
""""

# convert predictions and filenames to numpy arrays
__filenames = np.array([_filenames])
_predictions = np.array(predictions)

# get a list where the predictions were correct, that is, the same class is found at the same index
in __filenames and _predictions arrays

correct = __filenames == _predictions

# count the number of correct preditions

n_correct = correct.sum()

n_correct
""""
1927
""""
```

```
test_accuracy = n_correct / n_test_images

test_accuracy
""""
0.789
""""
```

Justification

This project’s search for the best combination of CNN parameters and hyperparameters used a nested approach. Early hyperparameter optimization focused on parameters common to all training jobs: batch size and steps size were determined initially and their choice carried to the remaining experiments. It was determined that a reasonably sized batch size of 128, and a step size of ½ the training data allowed for the fastest training without performance degradation. Considering that this optimization took a very small fraction of the total training time, it provided great savings in time and $$ as it halved the training time required for each training job. Another evaluation early on was the effect of image augmentation. Consistent with previously reported results, this project also found no advantage of including image augmentation. This also represented a great cost saving, as image augmentation more than doubled the training time. Most of the project’s effort was dedicated to comparing model performance of all 70 combinations of ImageNet CNN architectures and optimizers available on Keras, with optimizer default parameters. Additionally, a limited hyperparameter optimization was performed with 2 CNNs (mobilenet, VGG19) and 2 optimizers (RMSprop, SGD). As suggested by Keras’ documentation, the default parameters are optimal, as no combination outperformed those.

I’ve found that more recent architectures, such as mobilenet and densenet outperform older architectures both in accuracy and time to train. Several optimizers perform well, but Adam stood out as the winner for this particular problem. This combination was able to maximize the validation accuracy at 92%. Fine tuning of the model was able to push the accuracy one more percentage point.

V. Conclusion

Free-Form Visualization

The following code picks 5 random images of 10 random flower classes to predict / plot. Figure 10 shows each image with the actual class / prediction in each image’s label. As can be seen, the prediction is correct for most images, with a few exceptions in the morning glory (1 error out of 5), bougainvillea ((1 error out of 5) and hibiscus classes (2 errors out of 5).

```
test_dir_path = "/data/oxford102/test/"

# choose 10 random flower classes
choice_classes = np.random.choice(os.listdir(test_dir_path), 10, replace=False)

# pick 5 random images of each class to predict
n_images = 5
l = [np.random.choice(os.listdir(os.path.join(test_dir_path, choice_classes[i])), n_images) for i in
range(len(choice_classes))]
flat_list = [item for sublist in l for item in sublist]

# define figure with rows x columns
h, w = 15, 48  # height and width in inches
fig = plt.figure(figsize=(h, w))
columns = 5
rows = 20

for i, image in enumerate(flat_list):
    image_class = math.floor(i / 5)
    df2 = results['Filename'] == choice_classes[image_class] + "/" + image
    prediction = results.iloc[np.argmax(df2, axis=1)]['Predictions']
    ax = fig.add_subplot(rows, columns, i + 1)
    ax.set_xlabel(choice_classes[image_class] + " / " + prediction)
    img = mpimg.imread("/data/oxford102/val/" + choice_classes[image_class] + "/" + image)
    imgplot = plt.imshow(img)

fig.tight_layout()
plt.show()
```

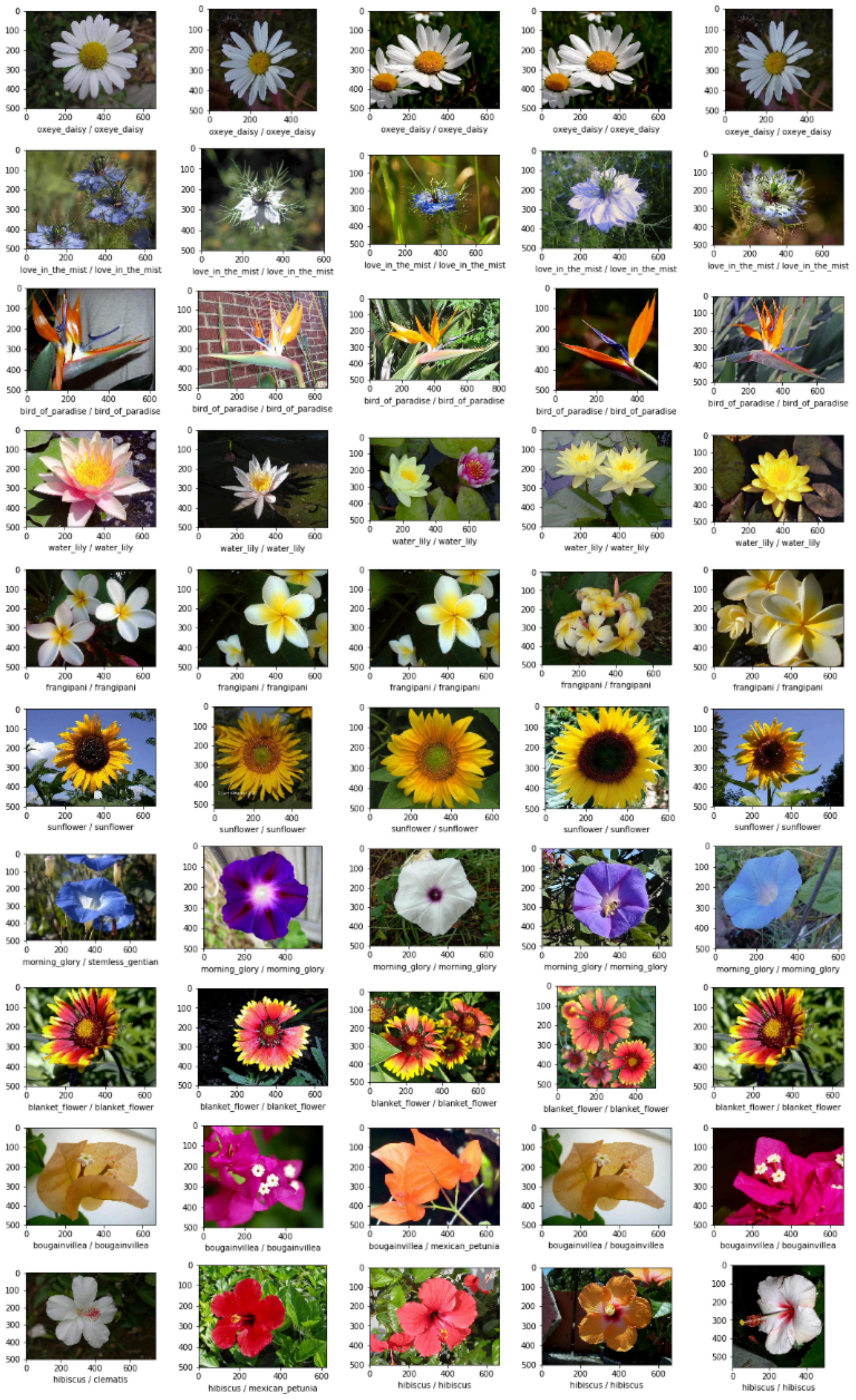

oxeye_daisy / oxeye_daisy
oxeye_daisy / oxeye_daisy
oxeye_daisy / oxeye_daisy
oxeye_daisy / oxeye_daisy
oxeye_daisy / oxeye_daisy
love_in_the_mist / love_in_the_mist
love_in_the_mist / love_in_the_mist
love_in_the_mist / love_in_the_mist
love_in_the_mist / love_in_the_mist
love_in_the_mist / love_in_the_mist
bird_of_paradise / bird_of_paradise
bird_of_paradise / bird_of_paradise
bird_of_paradise / bird_of_paradise
bird_of_paradise / bird_of_paradise
bird_of_paradise / bird_of_paradise
water_lily / water_lily
water_lily / water_lily
water_lily / water_lily
water_lily / water_lily
water_lily / water_lily
frangipani / frangipani
frangipani / frangipani
frangipani / frangipani
frangipani / frangipani
frangipani / frangipani
sunflower / sunflower
sunflower / sunflower
sunflower / sunflower
sunflower / sunflower
sunflower / sunflower
morning_glory / stemless_gentian
morning_glory / morning_glory
morning_glory / morning_glory
morning_glory / morning_glory
morning_glory / morning_glory
blanket_flower / blanket_flower
blanket_flower / blanket_flower
blanket_flower / blanket_flower
blanket_flower / blanket_flower
blanket_flower / blanket_flower
bougainvillea / bougainvillea
bougainvillea / bougainvillea
bougainvillea / mexican_petunia
bougainvillea / bougainvillea
bougainvillea / bougainvillea
hibiscus / clematis
hibiscus / mexican_petunia
hibiscus / hibiscus
hibiscus / hibiscus
hibiscus / hibiscus

Fig. 10 – Class prediction of 50 images from 5 flower classes. Labels are true/predicted class

Reflection

This Project aimed to evaluate the multiple options available in Keras for training a CNN for image classification problems using transfer learning, in which a network previously trained for a different task is used, compared to a simple benchmark model based on a few convolutional layers. It trained a classifier to recognize common UK flowers, taking advantage of the oxford102 dataset. The main goal was to evaluate the balance between accuracy and training time (a proxy for cost). The whole project, from idea to output, involved several steps and a very long process. It started with collecting the data, which fortunately was readily available online as a labeled dataset; writing code to handle the AWS infrastructure for model training; write a large number of experiments in Jupyter notebooks to optimize parameters; compare the results of experiments; and produce a report. Most of the time was spent evaluating and comparing different architectures, different optimizers, and hyperparameters. In the end, large difference in performance from the older to the newer network architectures was observed. Keras' suggestion to avoid wasting time with optimizer hyperparameter tuning was correct. It also stood out that batch size and step size optimization was quick and simple to do, and carried its benefits throughout the remaining of the project. It is also obvious that a major limitation of classifiers is the quality of the dataset. The best possible validation accuracy obtained overall was 93%, which is expected given the heterogeneity of the images. This project has highlighted a path to quickly select the best architecture and hyperparameters in the future. This exercise could be very valuable as a guide to classifier training with larger datasets and additional data processing required. The lessons from this project will carry to other image classification problems, facilitating the choice of base parameters and focus on more granular, problem specific research.

Evaluating all options was a laborious and time consuming project. In the future, optimizations could be performed on a subset of the problem. For instance, instead of classifying 102 image classes, the problem could be turned into a binary classification problem, since the image features would be the same (taking care of class imbalance).

Another major conclusion is that the price tag for training such a classifier justifies the purchase of our very own equivalent GPU, such as a NVidia 2080 card.

Improvement

The best classifier achieved in this project based on transfer learning of Keras supported ImageNet architectures, after extensive hyperparameter optimization and fine tuning, has validation and test accuracy of 0.93 and 0.79, respectively. The models are all a little bit overfit, as the accuracy vs epoch curves show consistently higher scores for training than the validation datasets. Algorithmic approaches for regularization have provided only small improvements. Including more data through image augmentation did not increase the model performance. One potential path to improve the model could be to collect more and better data. Since this implies a significant effort, especially considering that the dataset was collected by a 3rd party, adding new images might not be feasible. Another potential path might be new algorithmic strategies. For instance, one could use an ensemble of models, or a pipeline of models to make predictions. The images are very heterogenous in the nature of their content: some show a single plant while others show multiple plants (of the same class); some show mostly plant parts while others show mostly background; some show one part of the plant, e.g. a flower while others show either a distinct part of the plant (e.g. leaves), or even multiple distinct parts, e.g. flowers and leaves; and yet some show mostly background, and some focus mostly on the plant. One can hypothesize that this heterogeneity might be limiting the potential quality of the model. One could include other models as part of an ensemble or pipeline to identify parts of the plant, e.g. by image segmentation (of note, Oxford also providers a flower segmentation dataset), and this the classifier, or multiple classifiers would focus a specific plant part.

References:

1. https://arxiv.org/abs/1807.05284
2. https://www.researchgate.net/publication/317607639_Deep_learning_traffic_sign_detection_recognition_and_augmentation
3. http://yann.lecun.com/exdb/lenet/
4. http://people.idsia.ch/~juergen/computer-vision-contests-won-by-gpu-cnns.html
5. https://arxiv.org/abs/1811.09030

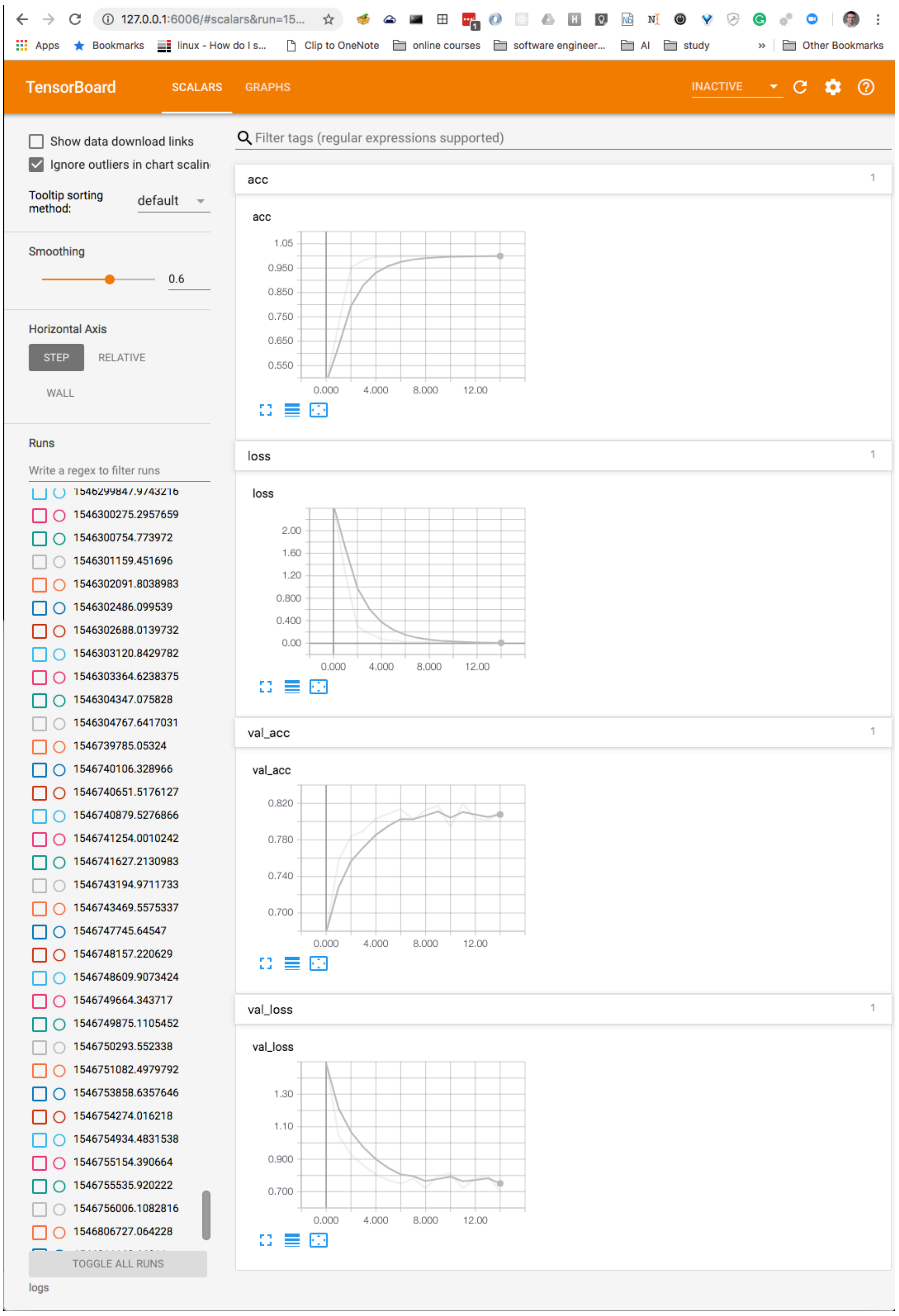
Appendix 1: Data Sources

The oxford102 category flower dataset can be downloaded from:
http://www.robots.ox.ac.uk/~vgg/data/flowers/102/102flowers.tgz

The original image labels can be downloaded from:
http://www.robots.ox.ac.uk/~vgg/data/flowers/102/imagelabels.mat

Actual flower class labels can be downloaded from:
https://raw.githubusercontent.com/jimgoo/caffe-oxford102/master/class_labels.py

Appendix 2: Example of tensorboard dashboard



Show data download links

Ignore outliers in chart scalin

Tooltip sorting method: default

Smoothing 0.6

Horizontal Axis: STEP RELATIVE WALL

Runs

Write a regex to filter runs

1546299847.9743216
1546300275.2957659
1546300754.773972
1546301159.451696
1546302091.8038983
1546302486.099539
1546302688.0139732
1546303120.8429782
1546303364.6238375
1546304347.075828
1546304767.6417031
1546739785.05324
1546740106.328966
1546740651.5176127
1546740879.5276866
1546741254.0010242
1546741627.2130983
1546743194.9711733
1546743469.5575337
1546747745.64547
1546748157.220629
1546748609.9073424
1546749664.343717
1546749875.1105452
1546750293.552338
1546751082.4979792
1546753858.6357646
1546754274.016218
1546754934.4831538
1546755154.390664
1546755535.920222
1546756006.1082816
1546806727.064228