# Convolutional Neural Network Implementation on Databricks

Evan Eames

## Contents

## 1 Motivation

Convolutional Neural Networks (CNN) are state-of-the-art Neural Network architectures which are primarily used for computer vision tasks. CNN can be applied to a number of different tasks, such as object classification (also known as image recognition), object localization, image segmentation, change detection, as well as similar tasks applied to video footage.

This document serves to outline how to implement a Residual CNN (often abbreviated Res-CNN, or simply ResNet), as well as how to migrate the network to the cloud (specifically via Databricks). The files for an example ResCNN project can be downloaded here:
https://github.com/EvanEames/Cars
Note that much of this code came from a previous respository, found here:
https://github.com/foamliu/Car-Recognition
For technical documentation on the ResCNN architecture, see:
https://arxiv.org/pdf/1512.03385.pdf

## 2 Implementation

### 2.1 Preliminaries

When you break apart a CNN, they comprise of different 'blocks', with each block simply representing a group of operations to be applied to some input data. These blocks can be broadly categorized into:
Identity Block: A series of operations which keep the shape of the data the same.
Convolution Block: A series of operations which reduce the shape of the input data to a **smaller** shape[1].
A CNN is a series of both Identity Blocks and Convolution Blocks (or Conv Blocks) which reduce an input image to a compact group of numbers. Each of these resulting numbers (if trained correctly) should eventually tell you something useful towards classifying the image.

A Residual CNN adds an addition step for each block. The data is saved as a temporary variable before the operations that constitute the block are applied, and then this temporary data is added to the output data. Generally, this additional step is applied to each block.

[1]For some processes, such as image reconstruction, or super-resolution applications, one may require blocks which increase the shape of the data. These are sometimes called 'Deconvolution Blocks'.

To visualize this, imagine a series of $n$ blocks as being represented by functions $f_1 \ldots f_n$, and imagine we begin with some input data $X$. Then a normal CNN is simply repeated application of $f_3(f_2(f_1(X)))$ and so on (until $f_n$, the final block, is applied). We will finally arrive at some $Y_n$, which is the final output of simplified data.
A Res-CNN, however, would begin with:
$f_1(X) + X = Y_1$ (where $Y_1$ is the output of the first block).
We then continue this:
$f_2(Y_1) + Y_1 = Y_2$
$f_3(Y_2) + Y_2 = Y_3$
...
$f_n(Y_{n-1}) + Y_{n-1} = Y_n$
The reasons for which these 'residual blocks' are more effective is debated, however, whatever the reason, these networks seem to excel at computer vision tasks. Also note that each function $f$, representing a block (that being a series of operations), is not necessary an individual conv block or identity block. They could also be combinations of both. An example image of a ResNet is shown in figure 1.

### 2.2 Keras Implementation

There are many different methods of implementing a Neural Network. One of the more intuitive ways is via Keras. Keras provides a simple front-end library for executing the individual steps which comprise a neural network. Keras can be configured to work with a Tensorflow back-end, or a Theano back-end. Here, we will be using a Tensorflow back-end. A Keras program is broken up into multiple parts. In general there is a 'main file' in which parameter values are set and functions are called, as well as other files which contain the architecture of the CNN, functions, data reading options, etc.

In the example ResNet provided in https://github.com/EvanEames/Cars, the files are as follows:

- CarModel.py - The main file.
- res_net.py - Contains the primary structure of the ResNet, in terms of Convolution Blocks and Identity Blocks (to be read in by the CarModel.py file).
- conv_block.py - Contains the structure of a Convolution Block (to be read in by the res_net.py file)

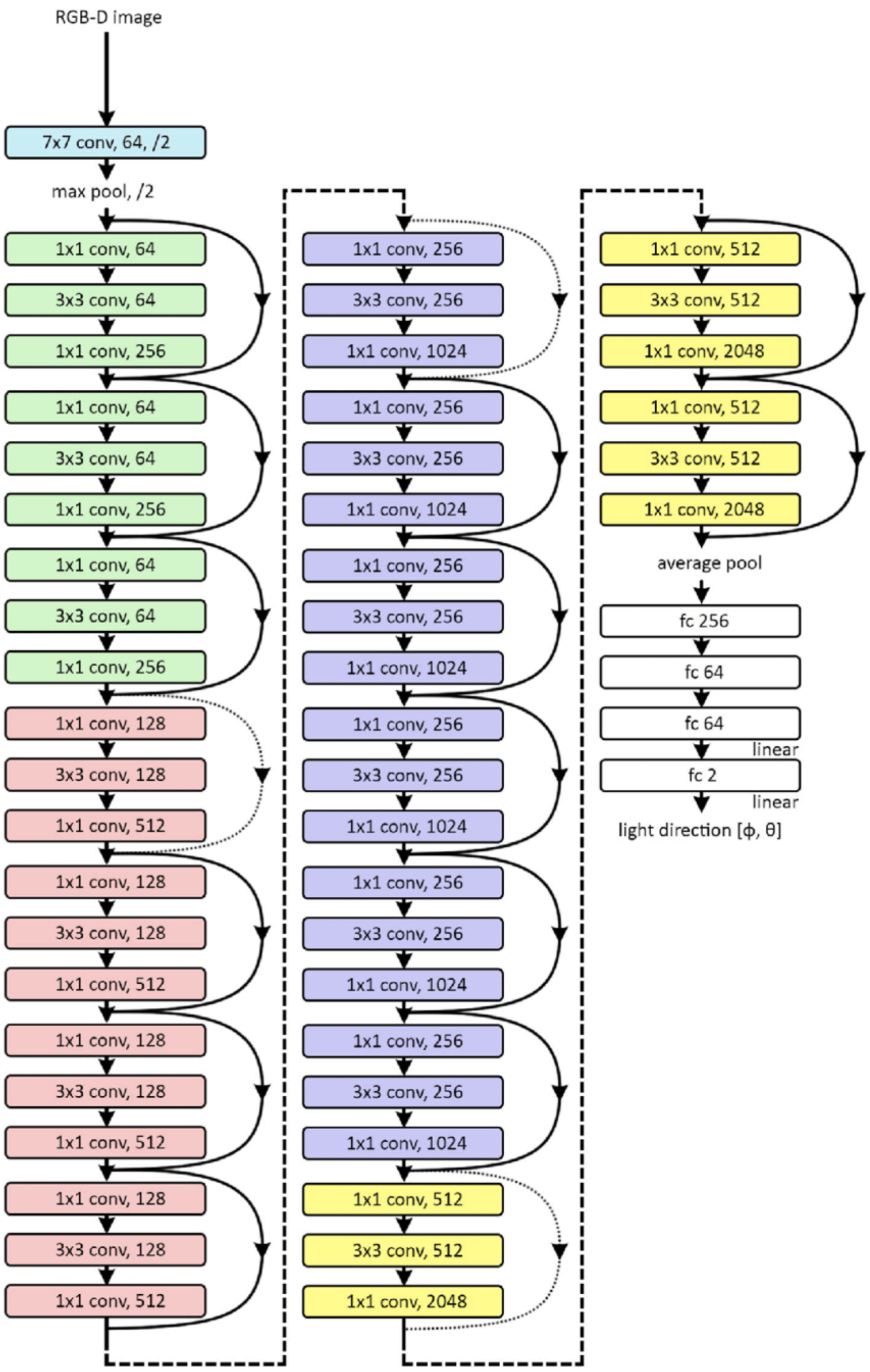


Figure 1: An example of a Residual CNN. The arrows jumping over various blocks represent the 'skip' connections characteristic of a Residual CNN.

- identity_block.py - Contains the structure of an Identity Block (to be read in by the res_net.py file)
- resnets_utils.py - Contains various helper functions, such as those for reading and formatting data.

Let's walk through the contents of the CarModel.py file.

**Part 0:** At the very beginning, we import all the necessary functions[2] and clear the trash. Then we set the preliminaries:

- img_width
- img_height
- classes (the number of objects we want to be able to identify)
- epochs (how many times we want the neural network to go through the training data)
- patience (if the accuracy is not improving, how many epochs we want the neural network to try to improve before giving up)
- verbose (program will output updates as it trains if this is set to 1)
- num_train_samples (how much of the training data should be used towards actually training the data, as opposed to cross validation)
- num_valid_samples (how much of the training data should be used towards cross validation, usually around 20% of the total training data)
- mode (what optimizer should be used in trying to minimize the training error)

**Part 1:** At this point we need to initialize the neural network architecture to be used. This is done by:

1. Reading in the model defined in the res_net.py file
2. Writing the ResNet function (defined in the res_net.py file) to a variable called 'model'

[2]If you have not used Keras before, it is quite possible you will need to download some of the packages using pip.

3. Deciding on which optimizer to use (lr = learning rate, decay = how much to reduce the lr each epoch, beta/epsilon are all hyperparameters)
4. Load pre-trained weights if required
5. Finally, compile the model via the command: model.compile(optimizer=optimizer, loss='categorical_crossentropy', metrics=['accuracy'])

**Part 2:** Now we can set up what are called callbacks. These are handy little pre-built additions that can either help the neural network train better, or track the training process. A few common ones I use include:

- ModelCheckpoint - Probably the most important. This saves the weights at the end of each run, and then you can read them in later to train further without starting from scratch.
- CSVLogger - This outputs the accuracy and cross validation (val) accuracy after each epoch. You can also plot this (see section 3.7).
- EarlyStopping - Stops the training if it seems as though accuracy has reached a maximum.
- ReduceLROnPlateau - Reduces learning rate when the accuracy reaches a 'plateau'. That is to say, when it becomes a bit harder to learn, the network changes more gradually so as to more carefully find what changes improve performance.

There are lots of other callbacks. You can see the full list here:
https://keras.io/callbacks/

**Part 3:** Train the model. The network will now begin the process of forward propagation and back propagation. Depending on the complexity of the network, and size of the Neural Network, this can take minutes to hours (state-of-the-art networks are sometimes trained for days or weeks). In general, if this is taking more than an hour, it should really be moved to the cloud (see section 3). As the network trains, you will see the training accuracy being printed at each step. At the end of each epoch, the network will use the cross validation data (assuming you allotted some) to check the validation error (val_acc). This val_acc provides a much better estimate of what the final test accuracy will be than the train_acc, which should always be larger than the val_acc.

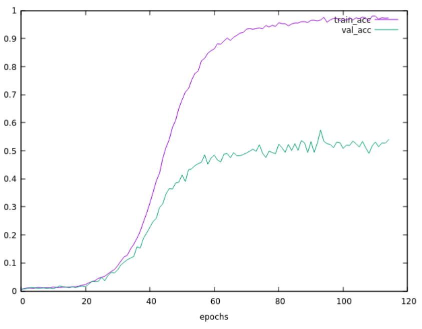
Also note that the weights file (which will be systematically saved if you have the ‘ModelCheckpoint’ callback) can be quite large, potentially up to a gigabyte.

### 2.2.1 Overfitting

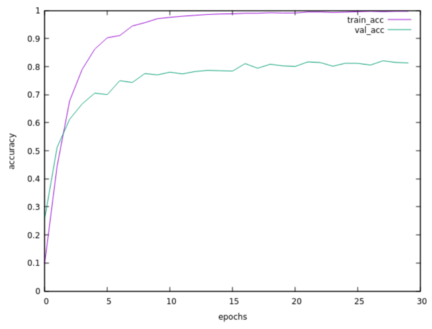
Note that if the difference between the train_acc and then val_acc (or, more-or-less equivalently, the difference between the train_acc and the test_acc) becomes too large, this is a case of overfitting. This means that the network has learned the training data *too* well, and is unable to generalize to new data. However, as one tries to push a neural network further and further, it generally happens that at high accuracy train_acc, the train_acc will eventually diverge from the val_acc. There is no hard-rule for when this divergence is concerning. It depends hugely on the type of network, the depth, the hyperparameters, the amount of input data, and the nature of the computer vision task.

However, as a rule-of-thumb, and speaking entirely from my personal experience, I can offer the following advice. It seems to me as though a good neural network should be able to keep train_acc and val_acc relatively paired until at least 70% accuracy, and when they do eventually diverge, this divergence shouldn’t exceed much more than 10%. As an example, figure 2 shows two test/validation accuracy curves. In figure 2 (a) we see that the train_acc and test_acc begin to diverge at extremely low accuracy (around 10 - 20%), and then the gap between the two becomes very large (around 50%). This is a case of fairly serious overfitting. Conversely, in figure 2 (b) we see that the two curves stay relatively well paired until around 60 - 70%, and then the difference between them remains under 20%. Again, the network is still overfitting, yet at such high accuracy it becomes increasingly difficult to do much better (even state-of-the-art implementations on the Standford Cars dataset begin overfitting above ∼88-93%, see Table 1 in https://arxiv.org/pdf/1702.01721.pdf).

**Part 4:** After training, the final step is to test the accuracy on some data that the Neural Network has not yet seen (test data). It is generally considered standard to use 50% of the total data set for testing, although this number can vary depending on the quantity of data. The test accuracy (test_acc) is considered the true metric for how well the Neural Network performs.



(a) A training curve exhibiting severe overfitting.



(b) A training curve exhibiting moderate overfitting.

Figure 2: Two example training curves.

## 2.3 Data Formatting

A huge part of any computer vision task is collecting, cleaning, and labelling the data. I could write an entire book here about the subject, and there is no clear 'best practice' for data preparation.

Instead, I will tell you how the data must be formatted so that it can be read in by Keras.

The Machine Learning world operates on HDF5 (Hierarchical Data Format 5) files (see https://en.wikipedia.org/wiki/Hierarchical_Data_Format). These are highly compressed files which are usually annotated by the .hdf5 or .h5 file extensions. Multiple images and labels can be written to HDF5 files, and then later extracted.

### 2.3.1 Reading an HDF5

Once data is saved to an HDF5, it can be later read through the following commands:

```
import h5py

train_dataset = h5py.File('training_data.h5', "r")

train_set_x_orig = np.array(train_dataset["train_set_x"][:])
train_set_y_orig = np.array(train_dataset["train_set_y"][:])
```

Here, train_set_x_orig will be a tensor containing all the images (features). If each image is 224×224×3, and you have $n$ images, the resulting tensor will have dimensions $224 \times 224 \times 3 \times n$. Conversely, train_set_y_orig is a 1D vector containing the corresponding labels.

### 2.3.2 Writing an HDF5

Creating an HDF5 file is somewhat more complicated. Yet in general one needs only a directory containing all the images, as well as a corresponding file containing all the labels. The images should be labelled sequentially (i.e. 00000.jpg, 00001.jpg, 00002.jpg, etc.), and the file containing the labels could be something like: [123, 17, 192, ...], where the first label (123) corresponds to the first image (00000.jpg), and so on. Note that it is also necessary to have a list somewhere which contains the correspondence between numerical labels and actual labels. For examples, 0 = dog, 2 = cat, 3 = horse, etc. These 'actual' labels are completely unnecessary for training and testing, but the code won't serve much good if, after lots of time spend training and testing, you don't know what real-world objects have actually been identified!

Some sample code for writing HDF5 files from an image directory and a file of corresponding labels can be found here:
https://github.com/EvanEames/h5_ReadWrite

**Note:** This is certainly not the only way to create HDF5 files, nor to format the data within them. I have, unfortunately, seen different teams using different conventions. For example, some teams prefer to first create one directory for each label, and then sort the images into the directories. When the HDF5 file is created, the label is inferred by the corresponding directory name. Other teams instead append a suffix to the file name, and infer the label from the file name upon reading. And within the HDF5 the images can also be stored as long 1D arrays. All this to say, be careful when dealing with somebody else's code. Run some sanity checks to make sure the train_set_x_orig and train_set_y_orig numpy arrays contain the data that you think they do.

## 2.4 Keras Blocks

Just in case you're curious about how all of this works, here is a little overview of the 'under-the-hood' components of a Keras Neural Network.

Inside the res_net.py file (as well as the conv_block.py and identity_block.py files) are a number of functions that correspond to various steps in a Neural Network. These steps combine together to make 'layers', and then a few layers will constitute a block. As an example of a single layer, we might have something like this:

```
X = Conv2D(some arguments)(X)
X = BatchNormalization(some arguments)(X)
X = Scale(some arguments)(X)
X = Activation('relu')(X)
```

This layer receives some input data[3] 'X' (likely an image) and consists of four steps. Firstly a convolution is applied to the data to make it slightly smaller. Secondly, Batch Normalization is applied (this is a step which both prevents overfitting, and also avoids a problem known as 'disappearing gradient', which is a bit more technical that I had planned on getting into). The third step relates to the fact that the pixel values of images are generally stored as integers, yet we need floats to allow for sufficient accuracy when performing backpropagation. Finally, the fourth step applies a Rectified Linear (ReLu) function (this is somewhat of a 'cut-off' to avoid negative activation values).

You may also notice that the conv blocks and identity blocks start with:

```
X_shortcut = X
```

and end with:

```
X = Add()([X,X_shortcut])
```

This is a 'skip connection', which is characteristic of Residual CNN (see figure 1). In the res_net.py file, there are a few other commands worth noting. For example, near the beginning you will find:

```
X = ZeroPadding2D((3, 3))(X_input)
```

This adds some extra pixels around the sides of the image (in this case 3 on each side), such that applying a convolution step (with a 7×7 filter[4]) will leave the image shape unchanged. Near the end we also have:

```
Xfc = AveragePooling2D(some arguments)(X)
```

[3]In Keras, the convention is that the function input is written in parentheses to the right of the function, e.g. X_out = Function(arguments)(X_in)

[4]Three pixels on each side, and one central pixel: thus 7.

This reduces the size of the data, similar to a convolution step, however instead of a filter in which pixels are multiplied, here we average across a region. Once we are finished with the convolution layers, we 'flatten' the data into a 1D vector. Finally we can include a 'fully connected', or 'dense' layer (a 1D layer in which all the nodes are fully connected to the next layer of nodes). These steps are accomplished via:

```
Xfc = Flatten()(Xfc)
Xfc = Dense(some arguments)(Xfc)
```

(and in case you're curious, 'Xfc' stands for 'X fully connected').

### 2.4.1 Transfer Learning

One technique often used to give a huge boost to performance is called 'Transfer Learning'. For this we read-in a set of weights that have been pre-trained on other data. These weights may already entail a concept of some basic structures (such as edges, corners, text, circles, etc.). This can save a significant amount of time, as a CNN will not have to 're-learn' these basic concepts (which is often the most time-consuming step).

In the res_net.py file we have applied transfer learning. The weights used are pre-trained on the ImageNet dataset[5], which contains 1000 object classes. As such, we first include a fully connected layer which maps to 1000 nodes. Then, after uploading the relevant weights, we switch this for a fully connected layer which maps to the number of nodes we need for our specific task (196 in our example, corresponding to the 196 classes of cars). The full code to accomplish this is:

```
X = the output of some previous conv and identity blocks

Xfc = AveragePooling2D((7, 7), name='avg_pool')(X)
Xfc = Flatten()(Xfc)
Xfc = Dense(1000, activation='softmax')(Xfc)

model = Model(inputs = X_input, outputs = Xfc, name='ImageNet1000')
model.load_weights('models/resnet152_weights_tf.h5', by_name=True)

Xfc2 = AveragePooling2D((7, 7), name='avg_pool')(X)
Xfc2 = Flatten()(Xfc2)
Xfc2 = Dense(196, activation='softmax')(Xfc2)
```

[5]https://en.wikipedia.org/wiki/ImageNet

```
model = Model(inputs = X_input, outputs = Xfc2, name='ResNet150')
```

To walk through this explicitly, we first take some output (X) from a number of conv and identity blocks. Then we map this to a layer consisting of a 1D layer of 1000 nodes (consistent with the number of object classes in the ImageNet dataset). We then define this Neural Network architecture as 'model', and load in weights (named here resnet152_weights_tf.h5) for this entire Neural Network (via the model.load_weights command). Lastly, we switch the last few layers (that conclude in a choice of 1000 classes) to the number of classes we need for our purposes (196 here, for the 196 different car models). Note how, in the fourth-to-last line we, input 'X' as opposed to Xfc (hence 'switching' the output layers for the new ones, denoted as Xfc2).

As a final step, this new model is saved to the 'model' variable (overwriting the previous one), however the values of the weights remain the trained values. Now this model can be called by the main program, for training and testing.

## 3 Running a Neural Network on Databricks

### 3.1 Logging into Databricks

You will need to first to login. Assuming your account has already been made, there should be a URL to follow, which looks something like this:
https://dbc-477ea27c-fb95.cloud.databricks.com

### 3.2 Setting up a Cluster

You can view the clusters by clicking 'clusters' on the left sidebar. You will probably not have permission to access other people's clusters, however you should be able to create your own (if the 'Create Cluster' button cannot be clicked, as the account admin for access).

Name and set-up your cluster. Be sure to choose one of the Runtime Versions that include 'ML' in the name (5.5 ML for example). Enable Autoscaling. The fastest configuration I was able to find was:

| | |
|---|---|
| Runtime Version | 5.5 ML GPU |
| Worker Type | p2.xlarge (Min 2, Max 8) |
| Driver Type | p3.16xlarge |

This resulted in a speed of about 150s/epoch using a 150 layer ResNet on the Standford Cars dataset with the sgd optimizer[6]. However, keep in

[6]lr=1e-3, decay=1e-6, momentum=0.9, nesterov=True

mind this is one of the more expensive clusters in terms of DBU (see section 3.6).

### 3.2.1 Libraries

The Runtime Versions with ML in their names should come with all relevant Machine Learning packages pre-installed (Keras, Tensorflow, openCV, etc.). If, for whatever reason, you need to use another Runtime Version (e.g. sometimes Databricks puts limits on how many clusters you can have running of the same instance type), then you can still install these libraries manually. To do this:

1. Click on the 'Databricks' button on the top of the left sidebar.
2. Under 'Common Tasks', click 'Import Library'.
3. For 'Library Source' choose 'PyPl'.
4. Ignore the Repository option, and simply type in the package name and click 'Create'.
5. You can choose to install libraries for all clusters, or only for select clusters.

In general, for the Computer Vision code included here, you should be fine with the following three libraries:

- keras
- tensorflow
- opencv-python

## 3.3 Moving Code into the Cloud

Your code must be adapted to Notebooks, which can then be run on databricks. Here you will find an example notebook, specifically dealing with an ML implementation using Keras:
https://docs.databricks.com/applications/deep-learning/single-node-training/keras.html

If you want to try this notebook yourself, or if you just want to use this Notebook as a blueprint for your own code, scroll down to the 'Example Notebook' section. On the right (in fairly small text) you will see 'Get notebook link'. You can click this and copy the Notebook link. Then switch back to Databricks, choose ‘Workspace’ on the left sidebar, choose your username, then click the little down-arrow next to your username, and choose ‘import’. Here you can paste the Notebook link, or you can upload your own files from the computer. To edit a cell within a Notebook, simply double-click it.

Before we upload the data (h5 files) we can first start re-touching the uploaded files (assuming you uploaded your own), to make sure they work on Databricks. In general, this shouldn’t be too much work, but here are a few considerations:

1. A few packages, such as those that deal with plotting, may not work in Databricks. Simply comment anything unnecessary.
2. To get a python script to read in another python script, you can generally use:

   ```
   from resnets_utils import *
   ```

   However, this won’t work on Databricks. In an otherwise empty cell, you can use:

   ```
   %run "/Users/<user>@datainsights.de/resnets_utils"
   ```

   **Note: It’s really important that this is the only command in the cell. Even adding comments and this will not work. So if you are reading in multiple other notebooks, you need one cell for each. I know it’s annoying, but I haven’t found another way to get it to work. Also, obviously all notebooks need to be in the same directory as the main notebook.**
3. For reading in data, you need to add the /dbfs/ prefix in front of the path name. As an example, when running my CNN on my computer, I used:

   ```
   train_dataset = h5py.File('datasets/train_cars.h5', "r")
   ```

   However, on Databricks this becomes:

   ```
   train_dataset = h5py.File('/dbfs/train_cars.h5', "r")
   ```

   The same is true for any callbacks (for example the path to the weights file or log file). You can also specify sub-directories within the /dbfs/ directory.

**Note:** If, on Databricks, you want to specifically play with the car identification example I have been working with, you can do so by importing the Cars.ipynb file, found here: https://github.com/EvanEames/Cars, to Databricks.

## 3.4 Uploading Data

The user interface (UI) that comes with Databricks doesn't seem to have an easy way to load in certain non-table files, specifically h5 files. However, you can do it via the Command Line Interface (CLI). Here are the steps:

1. First, in your terminal, install the Databricks CLI with:

   ```
   sudo pip install databricks-cli
   ```

2. Next, type:

   ```
   databricks configure --token
   ```

3. For the Databricks Host, enter the URL for your shared account (for example, https://dbc-477ea27c-fb95.cloud.databricks.com)
4. In the very top right of the Databricks window you will see a small person icon, which you can click to access the 'User Settings'
5. Switch to the 'Access Tokens' tab at the top, and then click 'Generate New Token'
6. Copy this in your terminal when you are prompted for the Token.
7. Now you will have access to Databricks directly through the CLI. You can see a list of functionalities with:

   ```
   databricks -h
   ```

   You can also see more specific commands by including the topic you want to know about, such as:

   ```
   databricks clusters -h
   ```

8. The command to upload a file is:

```
dbfs cp file.h5 dbfs:/file.h5
```

You can check what's on the dbfs with:

```
dbfs ls
```

Uploads can take some time, but when you finish uploading all the necessary files, you're good to get started!

## 3.5 Running Everything

To run your application, go to the clusters menu and start the one you wish to test on. Then go to the main Notebook, and on the top you will have an option to 'attach' the Notebook to one of the clusters. Note: you don't need to attach other Notebooks that are read/run by the main Notebook, only the main one.

It will take some time for your cluster to start running (it can be a bit long for larger ones). Once it is running, you can click 'Run All' at the top of your main Notebook (or run cells one-by-one for testing purposes with shift+enter).

**Note: Callbacks in Keras**
If your Machine Learning code makes use of callbacks, note they sometimes work a little differently in Keras. For example, the csv_logger callback is only printed to the /dbfs/ directory at the very end, not at each epoch (don't worry though, the log file will still contain the accuracy and val_accuracy for every epoch). One other important point is that **the weights file can take a long time to save to the /dbfs/ directory**, especially for deeper networks (30+ minutes is not unusual). Unfortunately, as far as I know, there isn't a way to terminate the cluster while this file saves.

## 3.6 Costs and Optimization

Databricks measures costs in 'DBU'. When you are configuring a cluster, the cost for given cluster types and sizes are listed (these are the costs in DBU per hour the cluster is active).

**Note: When you use multiple worker nodes, the DBU cost is multiplied by the number of nodes, so be careful with this.**

You will probably want to fiddle around a bit with different settings in terms of both your hyperparameters, as well as with the cluster type.

### 3.7 After Finishing

Once the training is finished, the weights are saved to the /dbfs/ directory, and the test accuracy has been calculated, you can download the weights to your local directory from the cloud using the following command:

```
dbfs cp dbfs:/weights.best.hdf5 ./weights.best.hdf5
```

These can then be used to make predictions for new data. You can also execute a similar command to download your log file:

```
dbfs cp dbfs:/training.log ./training.log
```

If you wish, you can plot the training and validation curves using gnuplot. Here are the commands (which must obviously be executed in gnuplot, in the same directory as the log file):

```
gnuplot> set datafile separator ","
gnuplot> set xlabel 'epochs'
gnuplot> set ylabel 'accuracy'
gnuplot> plot "training.log" u 1:2 with lines title 'train\_acc',
"training.log" u 1:4 with lines title 'val\_acc'
```

## 4 Deploying

Once the weights have been trained, and the test accuracy is satisfactory, it is time to deploy the network on the cloud. What we ultimately want is a program in which an image can be provided, and the output is the name of the object in the image. This could be done in two ways: either the image could be directly uploaded, or the image can be linked to via a url.

If you look at the 'guess.py' file found here:
https://github.com/EvanEames/Cars
Then you will find an implementation which takes a url argument and outputs the name of the predicted car model.

### 4.1 Deploying on Databricks

In the same github repository, you will also find the guess.ipynb file. This is a notebook implementation of the same code. Here you will find a quick overview of the content of the program.

**Part 0:** Firstly, as we did with the main code, we begin by initializing packages, clearing memory, and setting Keras variables. Other notebooks (resnets_utils, scale_layer, res_net) are also read in.

**Part 1:** Next, we read in the model we previously defined in the res_net file, as well as the weights we have now trained up to high accuracy (which should be stored in the dbfs directory). We also read in a file named cars_meta.mat (which comes pre-packaged with the Stanford Cars dataset). This file contains the correspondence between the label numbers and the label names (e.g. 0 = 2012 Tesla Model S, 1 = 2012 BMW M3 coupe, etc.). The code to accomplish this looks as follows:

```
img_width, img_height = 224, 224
model = load_model_weights()
model.load_weights('/dbfs/evan/weights.best.hdf5')
cars_meta = scipy.io.loadmat('/dbfs/evan/cars_meta')
class_names = cars_meta['class_names']  # shape=(1, 196)
class_names = np.transpose(class_names)
```

Note that the 'load_model_weights' function is defined in the resnets_utils file.

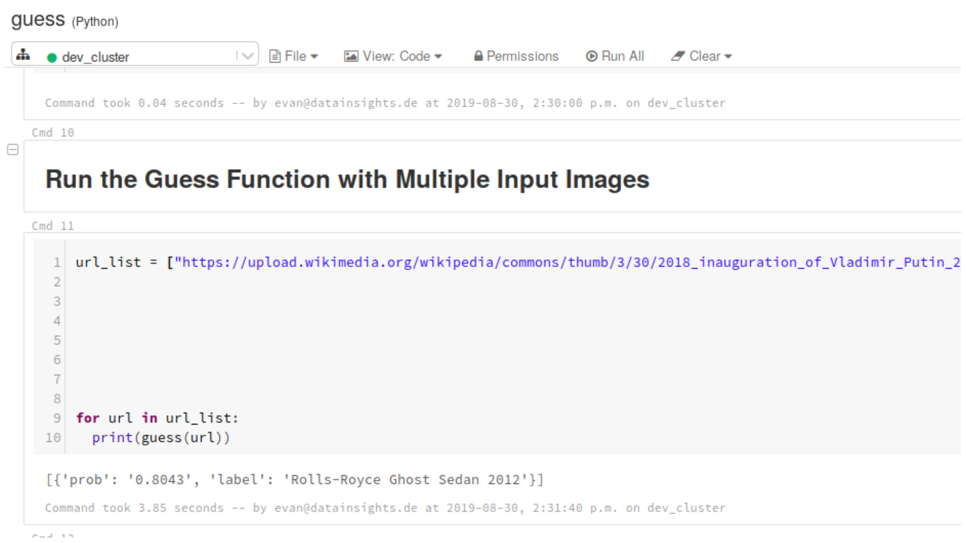
**Part 2:** Now, we can define a 'guess' function, which reads in a url to an image, and then outputs a prediction for the car model. At the core of this function are these three lines:

```
preds = model.predict(rgb_img)
prob = np.max(preds)
class_id = np.argmax(preds)
```

What we are doing here is inserting an RGB image into our model, and returning a 1D array (with 196 elements). The index which contains the highest value (defined as class_id here) is the 'best guess' for the object's class, and the value itself is a measure of how 'confident' the model feels about this guess.

**Part 3:** Now, we're free to play around with actual url links. The code should look something like this:

```
url_list = ["url_1", "url_2", "url_3", ... ]
for url in url_list:
  print(guess(url))
```



```
Command took 0.04 seconds -- by evan@datainsights.de at 2019-08-30, 2:30:00 p.m. on dev_cluster
```

Cmd 10

**Run the Guess Function with Multiple Input Images**

Cmd 11

```
url_list = ["https://upload.wikimedia.org/wikipedia/commons/thumb/3/30/2018_inauguration_of_Vladimir_Putin_2







for url in url_list:
  print(guess(url))
```

```
[{'prob': '0.8043', 'label': 'Rolls-Royce Ghost Sedan 2012'}]
Command took 3.85 seconds -- by evan@datainsights.de at 2019-08-30, 2:31:40 p.m. on dev_cluster
```

Figure 3: An example of Car Classification on a single picture (The state car of Vladimir Putin).

This code will return both a car model, and a corresponding confidence measure. You can see a real example (showing only this last cell in the notebook) in figure 3. Note that the output is shown at the very bottom, and correctly guesses the car as a Rolls-Royce Ghost Sedan 2012, with probability roughly 80%.

Note that if you're making a large amount of predictions at once, and want to save all your predictions in one place, you can write the output of the guess function to a 'results' variable, and then add this code:

```
with open('results.json', 'w') as file:
    json.dump(results, file, indent=4)
```

This will save predictions to a json file.

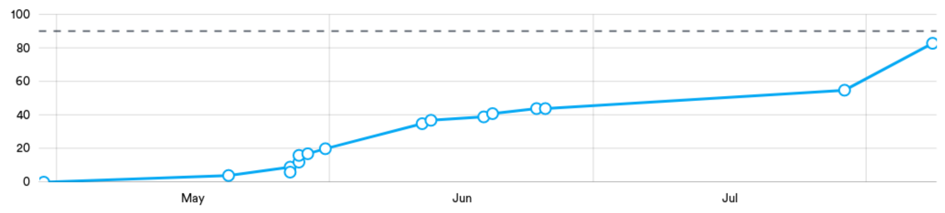


Figure 4: CNN Accuracy over time. I had set myself the personal goal of 90%, which I wasn't actually able to reach. I figured 84% was good enough to call it quits.

## 5 Closing Thoughts

I think this should be everything needed to implement an equally robust Deep Learning application. It took me a lot of toiling and fiddling to get it up to competitive (80%+ accuracy) levels. You can actually track my progress in figure 4.

Hopefully, if you use what I have presented here as a foundation from which to get started, the only tasks you will have to do are, firstly, data collection and organization, and, secondly, hyperparameter tuning. That should save you a huge amount of time.

So have fun exploring this exciting new world of Deep Learning! No making killer robots please...