TERM PROJECT: Quick, Draw! Doodle Recognition Challenge from Kaggle

CLASS: Neural Networks and Deep Learning with Python at Lawrence Technological University

AUTHOR: Yancong Nie

DATE: Dec 12, 2018

In [1]:
```
import keras
keras.__version__
```

```
Using TensorFlow backend.
```

Out[1]:
```
'2.2.4'
```

In [2]:
```
from google.colab import drive
drive.mount('/gdrive')
```

```
Drive already mounted at /gdrive; to attempt to forcibly remount, call drive.
mount("/gdrive", force_remount=True).
```

Quick, Draw

If you have a few minutes, go to the Quick Draw (https://quickdraw.withgoogle.com/#) website and have some fun. You have 20 seconds to draw a doodle of an object. An AI then tries to guess what you drew. This link Yancong's Quick Draw (https://drive.google.com/file/d/1TrvPQZuizY2uXde8wITdrYk_rfX3t-29/view?usp=sharing) is from my own drawing experiences.

Training Image Exploration

The simplified CSV training dataset that I use comes from the game Quick Draw. It contains 50M drawings encompassing 340 label categories. The values in the 'drawing' column contains a pair of normalised pixel position between 0 and 255. One for the x values in a stroke and one for the corresponding y values for the stroke. The length of an entry in 'drawing' corresponds to the number of strokes used in that drawing. Below shows an example of the simplified data. Code is adapted from https://www.kaggle.com/supers80/rvgis-exam/notebook (https://www.kaggle.com/supers80/rvgis-exam/notebook).

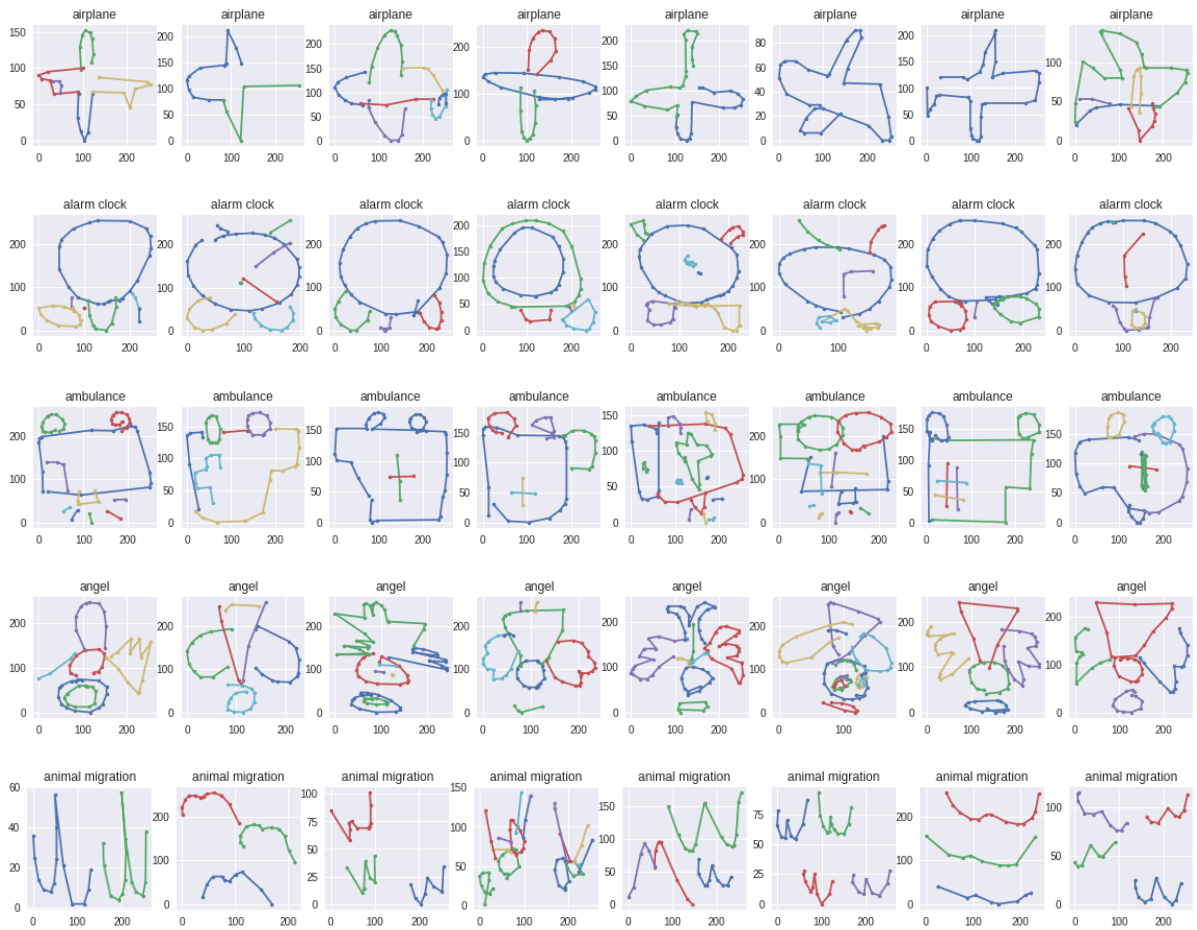
In [3]:
```
# Show simplified .csv data examples downloaded from kaggle to google drive
from glob import glob
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import ast

fnames = glob('/gdrive/My Drive/QuickDraw/train_simplified/*.csv')
cnames = ['countrycode', 'drawing', 'key_id', 'recognized', 'timestamp', 'wor
d']
drawlist = []
# Display the 5 first classes data in alphabetical order
for f in fnames[0:5]:
    first = pd.read_csv(f, nrows=20)
    # Make sure of getting 10 recognized drawings per class
    first = first[first.recognized==True].head(10)
    drawlist.append(first)
draw_df = pd.DataFrame(np.concatenate(drawlist), columns=cnames)
draw_df.head()
```

Out[3]:

	countrycode	drawing	key_id	recognized	timestamp	word
0	US	[[[167, 109, 80, 69, 58, 31, 57, 117, 99, 52, ...	5152802093400064	True	2017-03-08 21:12:07.266040	airplane
1	US	[[[90, 88, 95, 104, 112, 122], [65, 31, 12, 0,...	6577010312740864	True	2017-03-23 02:08:35.229980	airplane
2	US	[[[82, 49, 15, 4, 0, 5, 30, 85, 89, 93, 112, 1...	5643224746033152	True	2017-03-10 00:35:17.531970	airplane
3	IL	[[[64, 38, 23, 8, 0, 6, 26, 68], [74, 77, 84, ...	6670046841536512	True	2017-01-23 18:11:11.658170	airplane
4	US	[[[195, 164, 127, 40, 13, 0, 4, 28, 93, 172, 2...	5509429904539648	True	2017-03-02 19:06:55.494650	airplane

In [4]:
```
# Plot the drawings from the above 5 first classes
labels = unique_classes = draw_df['word'].unique()
for label in labels:
    plt.figure(figsize=(20,20))
    # Show 8 images for each class
    for ii in range(1,9):
        examples = [ast.literal_eval(pts) for pts in draw_df[draw_df['word']==
label].drawing.values]
        for x,y in examples[ii]:
            plt.subplot(8,8,ii)
            plt.plot(x,y,marker='.')
            # Comment out below line so that the drawings show in 255*255 axis
            # plt.axis('off')
            plt.title(label)
plt.show()
```

```
/usr/local/lib/python3.6/dist-packages/matplotlib/cbook/deprecation.py:106: M
atplotlibDeprecationWarning: Adding an axes using the same arguments as a pre
vious axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to eac
h axes instance.
  warnings.warn(message, mplDeprecation, stacklevel=1)
```



In [5]:
```
# Load the 50 first classes distribution to see how large each class is..
fnames = glob('/gdrive/My Drive/QuickDraw/train_simplified/*.csv')
cnames = ['countrycode', 'drawing', 'key_id', 'recognized', 'timestamp', 'wor
d']
drawlist = []
for f in fnames[0:50]:
    df = pd.read_csv(f)
    # Make sure of getting a recognized drawing
    df = df[df.recognized==True]
    drawlist.append(len(df))
plt.hist(drawlist,bins=15)
plt.title('Histogram of number of images for first 50 classes.')
plt.ylabel('Number of classes')
plt.xlabel('Number of images')
```

Out[5]: Text(0.5,0,'Number of images')

From the image distribution diagram, two obvious problems show as below, but both can be solved by limiting the number of images per class to be the same for all classes.

1. Just by judging the first 50 classes, most of the classes are between 100k-175k images. A dataset of that size will very quickly use up all the available memory.
2. It is not an equal distribution of images per class. This can cause overfitting to a few class since choosing the most represented classes will automatically increase accuracy during training because there are more instances of them.

Preparing the data

In [6]:
```
# Below vector to image data conversion is an adaptation from https://www.kaggle.com/supers80/rvgis-exam/notebook.
!pip install -q dask"[complete]"
import os
import re
from PIL import Image, ImageDraw
from tqdm import tqdm
from dask import bag
from sklearn.preprocessing import LabelEncoder
%matplotlib inline

classfiles = os.listdir('/gdrive/My Drive/QuickDraw/train_simplified')
numstonames = {i: v[:-4].replace(" ", "_") for i, v in enumerate(classfiles)}
# Adds underscores

num_classes = 10  #340 max
imheight, imwidth = 32, 32
ims_per_class = 500  # 1000 crashes colab easily. 500 images are able to fit in the memory together with the rest of the code

# Faster conversion function
def draw_it(strokes):
    image = Image.new("P", (256,256), color=255)
    image_draw = ImageDraw.Draw(image)
    for stroke in ast.literal_eval(strokes):
        for i in range(len(stroke[0])-1):
            image_draw.line([stroke[0][i],
                             stroke[1][i],
                             stroke[0][i+1],
                             stroke[1][i+1]],
                            fill=0, width=5)
    image = image.resize((imheight, imwidth))
    return np.array(image)/255.

# Workaround to load test data without crashing colab
test_df = pd.read_csv('/gdrive/My Drive/QuickDraw/test_simplified.csv', index_col=['key_id'])
image_series=test_df['drawing']
test_array = []
for row in range(0,len(image_series)):
    image = image_series.iloc[row]
    image = np.array(draw_it(image))
    image = np.reshape(image, image.shape + (1,))
    test_array.append(image)
test_df['image'] = test_array

# Get train arrays
train_grand = []
class_paths = glob('/gdrive/My Drive/QuickDraw/train_simplified/*.csv')
word_label = []
for i,c in enumerate(tqdm(class_paths[0: num_classes])):
    train = pd.read_csv(c, usecols=['drawing', 'recognized','word'], nrows=ims_per_class*5//4)
    # Make sure of getting a recognized drawing
    train = train[train.recognized == True].head(ims_per_class)
    word_label_current = train['word'].replace(' ', '_', regex=True)
```

```
    imagebag = bag.from_sequence(train.drawing.values).map(draw_it)
    trainarray = np.array(imagebag.compute())  # PARALLELIZE
    trainarray = np.reshape(trainarray, (ims_per_class, -1))
    labelarray = np.full((train.shape[0], 1), i)
    trainarray = np.concatenate((labelarray, trainarray), axis=1)
    train_grand.append(trainarray)
    word_label.append(word_label_current.values)

# Flatten the labels.
word_label = np.ravel(word_label)
word_encoder = LabelEncoder()
word_encoder.fit(word_label)

# Train_grand.pop() uses less memory than np.concatenate
train_grand = np.array([train_grand.pop() for i in np.arange(num_classes)])
train_grand = train_grand.reshape((-1, (imheight*imwidth+1)))
train_grand= np.append(train_grand, word_label[:,None],axis=1)

# Memory-friendly way to train/validation split
valfrac = 0.1
cutpt = int(valfrac * train_grand.shape[0])

np.random.shuffle(train_grand)
word_array = train_grand[:,-1]
train_grand = np.delete(train_grand,-1,axis=1)
train_grand = train_grand.astype(np.float64)
y_train, X_train = train_grand[cutpt: , 0], train_grand[cutpt: , 1:]
y_val, X_val = train_grand[0:cutpt, 0], train_grand[0:cutpt, 1:]

y_train_full_label = y_train
y_train = keras.utils.to_categorical(y_train, num_classes)
X_train = X_train.reshape(X_train.shape[0], imheight, imwidth, 1)
y_val_full_label = y_val
y_val = keras.utils.to_categorical(y_val, num_classes)
X_val = X_val.reshape(X_val.shape[0], imheight, imwidth, 1)

print(y_train.shape, "\n",
      X_train.shape, "\n",
      y_val.shape, "\n",
      X_val.shape)
```

```
100%|██████████| 10/10 [00:08<00:00,  1.15it/s]

(4500, 10)
 (4500, 32, 32, 1)
 (500, 10)
 (500, 32, 32, 1)
```

Reasoning for below final network:

1. All Keras pretrained convnet requires (150, 150, 3) input shape, not fit for quick draw's (32, 32, 1) shape.
2. Quick draw images are very similar to the handwriting digits in our textbook. So copied MNIST network as a start. Everything worked well, but there was overfitting problems. This link Yancong's Overfitting Problem (https://drive.google.com/file/d/1yhumtsU1_WggCZFR7rIvTjk8Tt2J2NBF/view?usp=sharing) shows more details.
3. To mitigate overfitting, added 2 Activation("relu") and 3 Dropout layers as below code shows:

In [0]:
```
#use a very basic CNN to classify the sketches(just added )
from keras import layers
from keras import models
from keras.layers import Dropout
from keras.layers.core import Activation

model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 1
)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(Activation("relu")) # Added to MNIST CNN to fight overfitting
model.add(Dropout(0.2))  # Added to MNIST CNN to fight overfitting
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(Activation("relu")) # Added to MNIST CNN to fight overfitting
model.add(Dropout(0.2)) # Added to MNIST CNN to fight overfitting
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(Dropout(0.5)) # Added to MNIST CNN to fight overfitting
model.add(layers.Dense(10, activation='softmax'))
```

In [17]:
```
# Display the model architecture
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_10 (Conv2D)           (None, 30, 30, 32)        320
_________________________________________________________________
max_pooling2d_7 (MaxPooling2 (None, 15, 15, 32)        0
_________________________________________________________________
activation_5 (Activation)    (None, 15, 15, 32)        0
_________________________________________________________________
dropout_7 (Dropout)          (None, 15, 15, 32)        0
_________________________________________________________________
conv2d_11 (Conv2D)           (None, 13, 13, 64)        18496
_________________________________________________________________
max_pooling2d_8 (MaxPooling2 (None, 6, 6, 64)          0
_________________________________________________________________
activation_6 (Activation)    (None, 6, 6, 64)          0
_________________________________________________________________
dropout_8 (Dropout)          (None, 6, 6, 64)          0
_________________________________________________________________
conv2d_12 (Conv2D)           (None, 4, 4, 64)          36928
_________________________________________________________________
flatten_4 (Flatten)          (None, 1024)              0
_________________________________________________________________
dense_7 (Dense)              (None, 64)                65600
_________________________________________________________________
dropout_9 (Dropout)          (None, 64)                0
_________________________________________________________________
dense_8 (Dense)              (None, 10)                650
=================================================================
Total params: 121,994
Trainable params: 121,994
Non-trainable params: 0
_________________________________________________________________
```

In [21]:
```
from keras.models import load_model
# Best model weights saved with the help of EarlyStopping)
model = load_model('/gdrive/My Drive/QuickDraw/project_perfect99.h5')

# Start training the model
from keras import optimizers
from keras.callbacks import EarlyStopping

model.compile(loss='categorical_crossentropy',
              optimizer=optimizers.RMSprop(lr=2e-5),
              metrics=['acc'])

# Using EarlyStopping to act on the model during training
early_stopping_monitor = EarlyStopping(patience=100)
history = model.fit(
      X_train,y_train,
      epochs=200, # Epochs over 2000 crashes colab easily
      batch_size=500, # Set the batch size based on images per class always wo
rks well.
      validation_data=(X_val,y_val),
      callbacks=[early_stopping_monitor])
```

```
Train on 4500 samples, validate on 500 samples
Epoch 1/200
4500/4500 [==============================] - 1s 312us/step - loss: 0.2087 - a
cc: 0.9256 - val_loss: 0.0445 - val_acc: 0.9900
Epoch 2/200
4500/4500 [==============================] - 0s 83us/step - loss: 0.2110 - ac
c: 0.9240 - val_loss: 0.0444 - val_acc: 0.9900
Epoch 3/200
4500/4500 [==============================] - 0s 75us/step - loss: 0.2035 - ac
c: 0.9264 - val_loss: 0.0446 - val_acc: 0.9900
Epoch 4/200
4500/4500 [==============================] - 0s 72us/step - loss: 0.1933 - ac
c: 0.9338 - val_loss: 0.0455 - val_acc: 0.9880
Epoch 5/200
4500/4500 [==============================] - 0s 73us/step - loss: 0.1878 - ac
c: 0.9344 - val_loss: 0.0447 - val_acc: 0.9880
Epoch 6/200
4500/4500 [==============================] - 0s 72us/step - loss: 0.2073 - ac
c: 0.9278 - val_loss: 0.0452 - val_acc: 0.9880
Epoch 7/200
4500/4500 [==============================] - 0s 72us/step - loss: 0.1944 - ac
c: 0.9322 - val_loss: 0.0456 - val_acc: 0.9880
Epoch 8/200
4500/4500 [==============================] - 0s 72us/step - loss: 0.2116 - ac
c: 0.9273 - val_loss: 0.0457 - val_acc: 0.9880
Epoch 9/200
4500/4500 [==============================] - 0s 71us/step - loss: 0.1951 - ac
c: 0.9318 - val_loss: 0.0453 - val_acc: 0.9860
Epoch 10/200
4500/4500 [==============================] - 0s 69us/step - loss: 0.1961 - ac
c: 0.9356 - val_loss: 0.0447 - val_acc: 0.9860
Epoch 11/200
4500/4500 [==============================] - 0s 70us/step - loss: 0.2059 - ac
c: 0.9298 - val_loss: 0.0453 - val_acc: 0.9860
Epoch 12/200
4500/4500 [==============================] - 0s 69us/step - loss: 0.1923 - ac
c: 0.9311 - val_loss: 0.0448 - val_acc: 0.9860
Epoch 13/200
4500/4500 [==============================] - 0s 69us/step - loss: 0.2027 - ac
c: 0.9309 - val_loss: 0.0456 - val_acc: 0.9860
Epoch 14/200
4500/4500 [==============================] - 0s 69us/step - loss: 0.1948 - ac
c: 0.9338 - val_loss: 0.0452 - val_acc: 0.9880
Epoch 15/200
4500/4500 [==============================] - 0s 70us/step - loss: 0.1924 - ac
c: 0.9324 - val_loss: 0.0453 - val_acc: 0.9880
Epoch 16/200
4500/4500 [==============================] - 0s 69us/step - loss: 0.2046 - ac
c: 0.9291 - val_loss: 0.0442 - val_acc: 0.9880
Epoch 17/200
4500/4500 [==============================] - 0s 69us/step - loss: 0.1844 - ac
c: 0.9353 - val_loss: 0.0444 - val_acc: 0.9880
Epoch 18/200
4500/4500 [==============================] - 0s 73us/step - loss: 0.1893 - ac
c: 0.9300 - val_loss: 0.0452 - val_acc: 0.9880
Epoch 19/200
4500/4500 [==============================] - 0s 71us/step - loss: 0.2008 - ac
```

```
c: 0.9296 - val_loss: 0.0469 - val_acc: 0.9880
Epoch 20/200
4500/4500 [==============================] - 0s 72us/step - loss: 0.1905 - ac
c: 0.9258 - val_loss: 0.0465 - val_acc: 0.9880
Epoch 21/200
4500/4500 [==============================] - 0s 73us/step - loss: 0.2148 - ac
c: 0.9227 - val_loss: 0.0474 - val_acc: 0.9880
Epoch 22/200
4500/4500 [==============================] - 0s 71us/step - loss: 0.2142 - ac
c: 0.9258 - val_loss: 0.0468 - val_acc: 0.9880
Epoch 23/200
4500/4500 [==============================] - 0s 72us/step - loss: 0.1856 - ac
c: 0.9356 - val_loss: 0.0458 - val_acc: 0.9880
Epoch 24/200
4500/4500 [==============================] - 0s 72us/step - loss: 0.1901 - ac
c: 0.9336 - val_loss: 0.0463 - val_acc: 0.9880
Epoch 25/200
4500/4500 [==============================] - 0s 71us/step - loss: 0.1969 - ac
c: 0.9333 - val_loss: 0.0470 - val_acc: 0.9860
Epoch 26/200
4500/4500 [==============================] - 0s 71us/step - loss: 0.1979 - ac
c: 0.9300 - val_loss: 0.0466 - val_acc: 0.9860
Epoch 27/200
4500/4500 [==============================] - 0s 71us/step - loss: 0.1899 - ac
c: 0.9353 - val_loss: 0.0476 - val_acc: 0.9880
Epoch 28/200
4500/4500 [==============================] - 0s 71us/step - loss: 0.1925 - ac
c: 0.9320 - val_loss: 0.0480 - val_acc: 0.9880
Epoch 29/200
4500/4500 [==============================] - 0s 72us/step - loss: 0.2032 - ac
c: 0.9278 - val_loss: 0.0471 - val_acc: 0.9880
Epoch 30/200
4500/4500 [==============================] - 0s 71us/step - loss: 0.1874 - ac
c: 0.9344 - val_loss: 0.0464 - val_acc: 0.9880
Epoch 31/200
4500/4500 [==============================] - 0s 72us/step - loss: 0.1955 - ac
c: 0.9320 - val_loss: 0.0468 - val_acc: 0.9880
Epoch 32/200
4500/4500 [==============================] - 0s 71us/step - loss: 0.2054 - ac
c: 0.9264 - val_loss: 0.0483 - val_acc: 0.9880
Epoch 33/200
4500/4500 [==============================] - 0s 72us/step - loss: 0.2098 - ac
c: 0.9282 - val_loss: 0.0463 - val_acc: 0.9900
Epoch 34/200
4500/4500 [==============================] - 0s 72us/step - loss: 0.2011 - ac
c: 0.9291 - val_loss: 0.0468 - val_acc: 0.9880
Epoch 35/200
4500/4500 [==============================] - 0s 72us/step - loss: 0.1925 - ac
c: 0.9329 - val_loss: 0.0457 - val_acc: 0.9880
Epoch 36/200
4500/4500 [==============================] - 0s 70us/step - loss: 0.1885 - ac
c: 0.9371 - val_loss: 0.0454 - val_acc: 0.9860
Epoch 37/200
4500/4500 [==============================] - 0s 70us/step - loss: 0.1912 - ac
c: 0.9291 - val_loss: 0.0474 - val_acc: 0.9860
Epoch 38/200
4500/4500 [==============================] - 0s 69us/step - loss: 0.2032 - ac
```

```
c: 0.9289 - val_loss: 0.0476 - val_acc: 0.9860
Epoch 39/200
4500/4500 [==============================] - 0s 70us/step - loss: 0.2074 - ac
c: 0.9251 - val_loss: 0.0492 - val_acc: 0.9880
Epoch 40/200
4500/4500 [==============================] - 0s 69us/step - loss: 0.1867 - ac
c: 0.9373 - val_loss: 0.0477 - val_acc: 0.9880
Epoch 41/200
4500/4500 [==============================] - 0s 71us/step - loss: 0.1904 - ac
c: 0.9344 - val_loss: 0.0486 - val_acc: 0.9840
Epoch 42/200
4500/4500 [==============================] - 0s 69us/step - loss: 0.1989 - ac
c: 0.9316 - val_loss: 0.0483 - val_acc: 0.9840
Epoch 43/200
4500/4500 [==============================] - 0s 69us/step - loss: 0.1968 - ac
c: 0.9307 - val_loss: 0.0480 - val_acc: 0.9840
Epoch 44/200
4500/4500 [==============================] - 0s 71us/step - loss: 0.1942 - ac
c: 0.9282 - val_loss: 0.0476 - val_acc: 0.9860
Epoch 45/200
4500/4500 [==============================] - 0s 71us/step - loss: 0.1860 - ac
c: 0.9380 - val_loss: 0.0471 - val_acc: 0.9840
Epoch 46/200
4500/4500 [==============================] - 0s 71us/step - loss: 0.1941 - ac
c: 0.9331 - val_loss: 0.0477 - val_acc: 0.9840
Epoch 47/200
4500/4500 [==============================] - 0s 73us/step - loss: 0.1890 - ac
c: 0.9344 - val_loss: 0.0472 - val_acc: 0.9860
Epoch 48/200
4500/4500 [==============================] - 0s 73us/step - loss: 0.1907 - ac
c: 0.9291 - val_loss: 0.0466 - val_acc: 0.9860
Epoch 49/200
4500/4500 [==============================] - 0s 70us/step - loss: 0.2034 - ac
c: 0.9262 - val_loss: 0.0482 - val_acc: 0.9860
Epoch 50/200
4500/4500 [==============================] - 0s 73us/step - loss: 0.2056 - ac
c: 0.9269 - val_loss: 0.0487 - val_acc: 0.9860
Epoch 51/200
4500/4500 [==============================] - 0s 73us/step - loss: 0.1933 - ac
c: 0.9351 - val_loss: 0.0475 - val_acc: 0.9880
Epoch 52/200
4500/4500 [==============================] - 0s 72us/step - loss: 0.1963 - ac
c: 0.9316 - val_loss: 0.0473 - val_acc: 0.9880
Epoch 53/200
4500/4500 [==============================] - 0s 72us/step - loss: 0.1982 - ac
c: 0.9316 - val_loss: 0.0470 - val_acc: 0.9880
Epoch 54/200
4500/4500 [==============================] - 0s 74us/step - loss: 0.1930 - ac
c: 0.9342 - val_loss: 0.0478 - val_acc: 0.9880
Epoch 55/200
4500/4500 [==============================] - 0s 72us/step - loss: 0.1872 - ac
c: 0.9331 - val_loss: 0.0475 - val_acc: 0.9860
Epoch 56/200
4500/4500 [==============================] - 0s 72us/step - loss: 0.1911 - ac
c: 0.9318 - val_loss: 0.0483 - val_acc: 0.9860
Epoch 57/200
4500/4500 [==============================] - 0s 74us/step - loss: 0.2107 - ac
```

```
c: 0.9238 - val_loss: 0.0488 - val_acc: 0.9840
Epoch 58/200
4500/4500 [==============================] - 0s 70us/step - loss: 0.1942 - ac
c: 0.9351 - val_loss: 0.0503 - val_acc: 0.9840
Epoch 59/200
4500/4500 [==============================] - 0s 71us/step - loss: 0.2075 - ac
c: 0.9287 - val_loss: 0.0499 - val_acc: 0.9840
Epoch 60/200
4500/4500 [==============================] - 0s 72us/step - loss: 0.2000 - ac
c: 0.9296 - val_loss: 0.0497 - val_acc: 0.9860
Epoch 61/200
4500/4500 [==============================] - 0s 70us/step - loss: 0.1986 - ac
c: 0.9309 - val_loss: 0.0485 - val_acc: 0.9860
Epoch 62/200
4500/4500 [==============================] - 0s 71us/step - loss: 0.1914 - ac
c: 0.9349 - val_loss: 0.0493 - val_acc: 0.9860
Epoch 63/200
4500/4500 [==============================] - 0s 73us/step - loss: 0.1964 - ac
c: 0.9291 - val_loss: 0.0492 - val_acc: 0.9860
Epoch 64/200
4500/4500 [==============================] - 0s 70us/step - loss: 0.2026 - ac
c: 0.9253 - val_loss: 0.0494 - val_acc: 0.9840
Epoch 65/200
4500/4500 [==============================] - 0s 69us/step - loss: 0.1952 - ac
c: 0.9293 - val_loss: 0.0500 - val_acc: 0.9860
Epoch 66/200
4500/4500 [==============================] - 0s 69us/step - loss: 0.1871 - ac
c: 0.9298 - val_loss: 0.0501 - val_acc: 0.9840
Epoch 67/200
4500/4500 [==============================] - 0s 72us/step - loss: 0.1945 - ac
c: 0.9353 - val_loss: 0.0490 - val_acc: 0.9840
Epoch 68/200
4500/4500 [==============================] - 0s 69us/step - loss: 0.1962 - ac
c: 0.9311 - val_loss: 0.0494 - val_acc: 0.9840
Epoch 69/200
4500/4500 [==============================] - 0s 69us/step - loss: 0.1883 - ac
c: 0.9342 - val_loss: 0.0499 - val_acc: 0.9840
Epoch 70/200
4500/4500 [==============================] - 0s 71us/step - loss: 0.2002 - ac
c: 0.9284 - val_loss: 0.0496 - val_acc: 0.9840
Epoch 71/200
4500/4500 [==============================] - 0s 69us/step - loss: 0.1879 - ac
c: 0.9347 - val_loss: 0.0502 - val_acc: 0.9840
Epoch 72/200
4500/4500 [==============================] - 0s 70us/step - loss: 0.1950 - ac
c: 0.9287 - val_loss: 0.0505 - val_acc: 0.9840
Epoch 73/200
4500/4500 [==============================] - 0s 74us/step - loss: 0.1961 - ac
c: 0.9329 - val_loss: 0.0516 - val_acc: 0.9840
Epoch 74/200
4500/4500 [==============================] - 0s 71us/step - loss: 0.1902 - ac
c: 0.9362 - val_loss: 0.0513 - val_acc: 0.9840
Epoch 75/200
4500/4500 [==============================] - 0s 71us/step - loss: 0.1885 - ac
c: 0.9367 - val_loss: 0.0506 - val_acc: 0.9840
Epoch 76/200
4500/4500 [==============================] - 0s 71us/step - loss: 0.1834 - ac
```

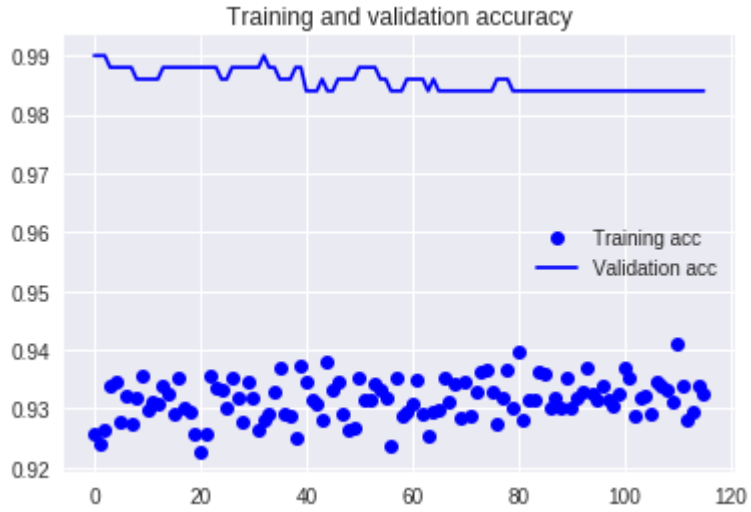
```
c: 0.9329 - val_loss: 0.0497 - val_acc: 0.9840
Epoch 77/200
4500/4500 [==============================] - 0s 72us/step - loss: 0.2054 - ac
c: 0.9273 - val_loss: 0.0499 - val_acc: 0.9860
Epoch 78/200
4500/4500 [==============================] - 0s 72us/step - loss: 0.1888 - ac
c: 0.9318 - val_loss: 0.0503 - val_acc: 0.9860
Epoch 79/200
4500/4500 [==============================] - 0s 74us/step - loss: 0.1846 - ac
c: 0.9364 - val_loss: 0.0501 - val_acc: 0.9860
Epoch 80/200
4500/4500 [==============================] - 0s 72us/step - loss: 0.1931 - ac
c: 0.9300 - val_loss: 0.0510 - val_acc: 0.9840
Epoch 81/200
4500/4500 [==============================] - 0s 72us/step - loss: 0.1750 - ac
c: 0.9396 - val_loss: 0.0513 - val_acc: 0.9840
Epoch 82/200
4500/4500 [==============================] - 0s 71us/step - loss: 0.1911 - ac
c: 0.9280 - val_loss: 0.0505 - val_acc: 0.9840
Epoch 83/200
4500/4500 [==============================] - 0s 72us/step - loss: 0.2013 - ac
c: 0.9313 - val_loss: 0.0503 - val_acc: 0.9840
Epoch 84/200
4500/4500 [==============================] - 0s 71us/step - loss: 0.1880 - ac
c: 0.9316 - val_loss: 0.0500 - val_acc: 0.9840
Epoch 85/200
4500/4500 [==============================] - 0s 72us/step - loss: 0.1876 - ac
c: 0.9362 - val_loss: 0.0492 - val_acc: 0.9840
Epoch 86/200
4500/4500 [==============================] - 0s 72us/step - loss: 0.1822 - ac
c: 0.9360 - val_loss: 0.0518 - val_acc: 0.9840
Epoch 87/200
4500/4500 [==============================] - 0s 71us/step - loss: 0.1972 - ac
c: 0.9300 - val_loss: 0.0508 - val_acc: 0.9840
Epoch 88/200
4500/4500 [==============================] - 0s 74us/step - loss: 0.1942 - ac
c: 0.9320 - val_loss: 0.0514 - val_acc: 0.9840
Epoch 89/200
4500/4500 [==============================] - 0s 72us/step - loss: 0.1931 - ac
c: 0.9302 - val_loss: 0.0519 - val_acc: 0.9840
Epoch 90/200
4500/4500 [==============================] - 0s 71us/step - loss: 0.1818 - ac
c: 0.9351 - val_loss: 0.0511 - val_acc: 0.9840
Epoch 91/200
4500/4500 [==============================] - 0s 72us/step - loss: 0.1916 - ac
c: 0.9300 - val_loss: 0.0513 - val_acc: 0.9840
Epoch 92/200
4500/4500 [==============================] - 0s 71us/step - loss: 0.1887 - ac
c: 0.9318 - val_loss: 0.0502 - val_acc: 0.9840
Epoch 93/200
4500/4500 [==============================] - 0s 68us/step - loss: 0.1885 - ac
c: 0.9329 - val_loss: 0.0511 - val_acc: 0.9840
Epoch 94/200
4500/4500 [==============================] - 0s 69us/step - loss: 0.1936 - ac
c: 0.9369 - val_loss: 0.0505 - val_acc: 0.9840
Epoch 95/200
4500/4500 [==============================] - 0s 68us/step - loss: 0.1908 - ac
```

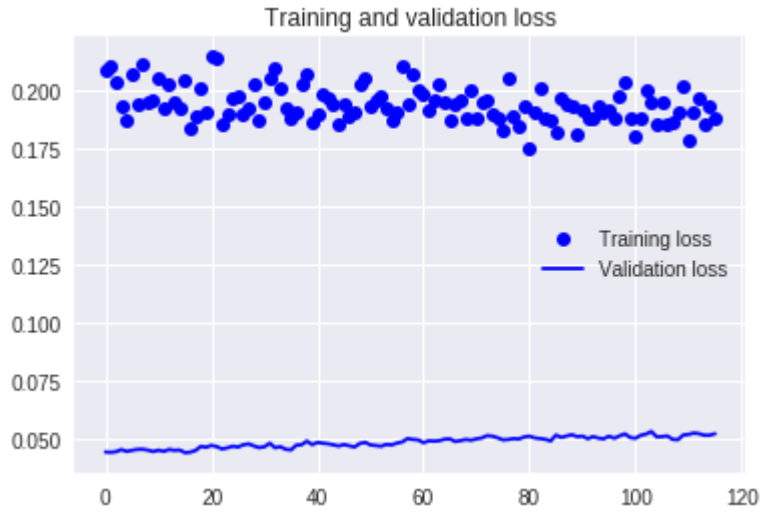
```
c: 0.9327 - val_loss: 0.0502 - val_acc: 0.9840
Epoch 96/200
4500/4500 [==============================] - 0s 71us/step - loss: 0.1919 - ac
c: 0.9313 - val_loss: 0.0513 - val_acc: 0.9840
Epoch 97/200
4500/4500 [==============================] - 0s 69us/step - loss: 0.1882 - ac
c: 0.9340 - val_loss: 0.0504 - val_acc: 0.9840
Epoch 98/200
4500/4500 [==============================] - 0s 69us/step - loss: 0.1974 - ac
c: 0.9316 - val_loss: 0.0514 - val_acc: 0.9840
Epoch 99/200
4500/4500 [==============================] - 0s 69us/step - loss: 0.2041 - ac
c: 0.9304 - val_loss: 0.0523 - val_acc: 0.9840
Epoch 100/200
4500/4500 [==============================] - 0s 70us/step - loss: 0.1882 - ac
c: 0.9324 - val_loss: 0.0507 - val_acc: 0.9840
Epoch 101/200
4500/4500 [==============================] - 0s 72us/step - loss: 0.1802 - ac
c: 0.9371 - val_loss: 0.0504 - val_acc: 0.9840
Epoch 102/200
4500/4500 [==============================] - 0s 73us/step - loss: 0.1887 - ac
c: 0.9353 - val_loss: 0.0517 - val_acc: 0.9840
Epoch 103/200
4500/4500 [==============================] - 0s 71us/step - loss: 0.2004 - ac
c: 0.9289 - val_loss: 0.0522 - val_acc: 0.9840
Epoch 104/200
4500/4500 [==============================] - 0s 73us/step - loss: 0.1952 - ac
c: 0.9318 - val_loss: 0.0533 - val_acc: 0.9840
Epoch 105/200
4500/4500 [==============================] - 0s 72us/step - loss: 0.1861 - ac
c: 0.9322 - val_loss: 0.0510 - val_acc: 0.9840
Epoch 106/200
4500/4500 [==============================] - 0s 71us/step - loss: 0.1952 - ac
c: 0.9291 - val_loss: 0.0511 - val_acc: 0.9840
Epoch 107/200
4500/4500 [==============================] - 0s 71us/step - loss: 0.1855 - ac
c: 0.9347 - val_loss: 0.0514 - val_acc: 0.9840
Epoch 108/200
4500/4500 [==============================] - 0s 71us/step - loss: 0.1868 - ac
c: 0.9340 - val_loss: 0.0500 - val_acc: 0.9840
Epoch 109/200
4500/4500 [==============================] - 0s 74us/step - loss: 0.1909 - ac
c: 0.9333 - val_loss: 0.0498 - val_acc: 0.9840
Epoch 110/200
4500/4500 [==============================] - 0s 73us/step - loss: 0.2018 - ac
c: 0.9311 - val_loss: 0.0518 - val_acc: 0.9840
Epoch 111/200
4500/4500 [==============================] - 0s 72us/step - loss: 0.1786 - ac
c: 0.9411 - val_loss: 0.0521 - val_acc: 0.9840
Epoch 112/200
4500/4500 [==============================] - 0s 73us/step - loss: 0.1911 - ac
c: 0.9338 - val_loss: 0.0527 - val_acc: 0.9840
Epoch 113/200
4500/4500 [==============================] - 0s 72us/step - loss: 0.1969 - ac
c: 0.9282 - val_loss: 0.0524 - val_acc: 0.9840
Epoch 114/200
4500/4500 [==============================] - 0s 71us/step - loss: 0.1856 - ac
```

```
c: 0.9293 - val_loss: 0.0517 - val_acc: 0.9840
Epoch 115/200
4500/4500 [==============================] - 0s 71us/step - loss: 0.1939 - ac
c: 0.9338 - val_loss: 0.0517 - val_acc: 0.9840
Epoch 116/200
4500/4500 [==============================] - 0s 71us/step - loss: 0.1880 - ac
c: 0.9327 - val_loss: 0.0524 - val_acc: 0.9840
```

In [0]:
```
model.save('/gdrive/My Drive/QuickDraw/project_perfect99.h5')
```

In [23]:
```
# Plot the results:
import matplotlib.pyplot as plt

acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(len(acc))

plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.legend()

plt.figure()

plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()

plt.show()
```





Research on why validation accuracy outperformed training accuracy?

My googled explanation from https://www.quora.com/How-can-I-explain-the-fact-that-test-accuracy-is-much-higher-than-train-accuracy (https://www.quora.com/How-can-I-explain-the-fact-that-test-accuracy-is-much-higher-than-train-accuracy) is:

1. Think of using Dropout as nearly identical to forcing your neural network to become a very large collection of weak classifiers (in other words, an ensemble). As the name implies, an individual weak classifier doesn't have much classification accuracy, they only become powerful once you string a bunch of them together.
2. Dropout, during training, slices off some random collection of these classifiers. Thus, training accuracy suffers.
3. Dropout, during testing, turns itself off and allows all of the 'weak classifiers' in the neural network to be used. Thus, testing accuracy improves.

This link Why Validation Accuracy Outperformed Training Accuracy (https://drive.google.com/file/d/1BKRYRonAQyEDHqOO3CJdB8byzvuDaNlh/view?usp=sharing) shows my varification on the above explanation. It seems reasonable.

Take Away from Quick Draw Kaggle Competition

1. Real world drawing dataset is huge, noisy and very expensive to convert and train. Most of the time, bigger GPU and RAM are needed. Otherwise, your system crashes easily.
2. All the training records and results should be saved timely into a safe place in case they got lost.
3. Save the trained model weights in a safe place is a great way to help you accumulate excellent training results and avoid many repeated training work.
4. Watch out the tradeoffs when using Dropout and Activation("relu") to mitigate overfitting.
5. Text book examples and knowledges are not enough to solve real world problems, you need do a lot of reseach on Google.
6. My pain with colab was that the connection died quite often, which made me lose my work and having to start all over.
7. Colab text cell doesn't show pictures and diagrams directly, instead you need to use hyperlink or powerpoint to assist for your presentation.
8. For a kaggle competition, you don't need to start from scratch, you can copy some of the trivial stuff like loading the data or submitting the data.
9. Overall I found the Quick Draw Kaggle Competition as an interesting and educational experience, and will attend similar ones in the future for sure.