Teach your Raspberry Pi – Episode 2
Multi-gesture recognition

Version 1.0

Teach your Raspberry Pi – Episode 2

Multi-gesture recognition

Release Information

Document History

Version	Date	Confidentiality	Change
1.0	25 April 2018	Non-confidential	First release

Non-Confidential Proprietary Notice

Confidentiality Status

Product Status

The information in this document is Final, that is for a developed product.

Web Address

http://www.arm.com

Contents

1 About this document

This document contains a how-to guide for implementing multi-gesture recognition using machine learning on a Raspberry Pi. You will download Arm's training scripts to assist you as you record your gestures with a camera, classify them, and create a neural network to perform real-time gesture recognition for control of your Pi.

2 Overview

This guide is the second episode in our series on how to teach your Raspberry Pi gesture recognition using a neural network.

Whereas Episode 1 shows you how to train a neural network on a Raspberry Pi to recognize one simple gesture, this guide shows you how to implement a neural network on your Pi that recognizes multiple gestures.

First, you will record a sequence of gestures and manually classify them. The footage is saved as a series of images files directly to a filesystem, which avoids loading to and from memory. These images becomes your training and validation data sets that you'll use to train a neural network from scratch which we can compare to previously trained models. We will introduce some techniques that help improve the performance of a neural network to perform gesture recognition using larger datasets and in different lighting conditions, and then explain how to deploy your new network with and integrate speech, music, and light control.

This demo video shows the resulting system that can be achieved Here you can see a user's gestures controlling a lamp, music playback and volume, triggering an audio speech file, and locking/unlocking a PC:

https://www.youtube.com/watch?v=JSFlpv93MdU

The neural network demonstrated in the video distinguishes between multiple gestures and works across varied lighting conditions and clothing. The neural network runs on a single Raspberry Pi 3 with a connected ZeroCam which is positioned in the corner of the room and out of shot.

The limitations of the simple approach used in Episode 1 were:

- Restricted data set size - all the images needed to be loaded into RAM before training.
- Limited sensitivity - the pretrained MobileNet was not well-suited to recognizing gestures that only use a few percent of the pixels or were not in the middle of the frame.

This episode addresses these limitations and outlines the process of designing and training neural networks for a specific task.

3 Prerequisites

Episode 1 introduced gesture recognition using a Raspberry Pi explained some of the fundamental concepts of machine learning. However, you can continue following this guide if you want to skip Episode 1.

This guide assumes some familiarity with Keras and neural network training. This Keras getting started guide gives an overview of the Keras Sequential model.

3.1. Install TensorFlow

For a Raspian base install the only dependency that you need to add is TensorFlow from Google's binaries. First, install some TensorFlow prerequisites by entering the following in the command line:

```
sudo apt-get install libblas-dev liblapack-dev python-dev libatlas-base-dev gfortran python-
setuptools python-h5py
```

The exact URL of the current TensorFlow build varies between versions. Go to the TensorFlow Raspberry Pi 3 build page to find the current one:

```
sudo pip2 install <insert .whl link from the build page here>
```

3.2. Install Arm's training scripts

Download or clone our ML examples repository from GitHub by entering the following on the command line:

```
git clone https://github.com/ARM-software/ML-examples.git
cd ML-examples/multi-gesture-recognition
```

These scripts are designed to be easy to understand and modify. Feel free to explore and hack them with your own changes.

Advanced information
The python source code is designed to be straightforward to follow:

- preview.py shows the current image from the camera to check it is pointing in the right direction.
- record.py captures images from the camera and saves them to disk.
- classify.py gives provides a basic GUI for labelling recorded images.
- merge.py combines classified images from multiple directories into one directory.
- validate_split.py seperates 10% of the images in a directory for use as a validation set.
- train.py loads trains a convolutional neural network to predict classes from images.
- test.py reports the accuracy of a trained classifier on a particular directory of images.
- run.py captures images from the camera, classifies them and prints the probabilities of each class.
- story.py is the script used to drive the live demo, turning lights on and off, playing music and executing remote commands. You will need to modify this to make it work in your environment.
- camera.py initializes the picamera module and optionally fluctuates the exposure and white balance during recording.

4 Record data

In Episode 1, images are converted to a list of numpy arrays, which are written to disk at the end of the program. This enables a higher frame rate on low-end Pi Zero hardware, but it limits the number of images that can be stored by the amount of RAM that is available. In this guide, we are going to write out each frame as a PNG (Portable Network Graphic) file.

4.1. Record video and classify actions

Start by setting up your filming environment. We recommend doing a test recording to ensure that your camera is positioned correctly to capture your actions.

Define your actions, or use the ones from our demo.

For our demo video, these classes were used:

Class	Action
0	None - moving around, sitting, putting on and taking off coat.
1	Door - coming in or leaving through the door
2	Light - pointing at the main light to turn it on/off
3	Music - holding both hands up to start the music playing
4	Stop - holding up one hand to dim the volume of the music

Rehearse a short sequence that encompasses all of the actions.

To film all of your gestures in one recording and write out the output as png files:

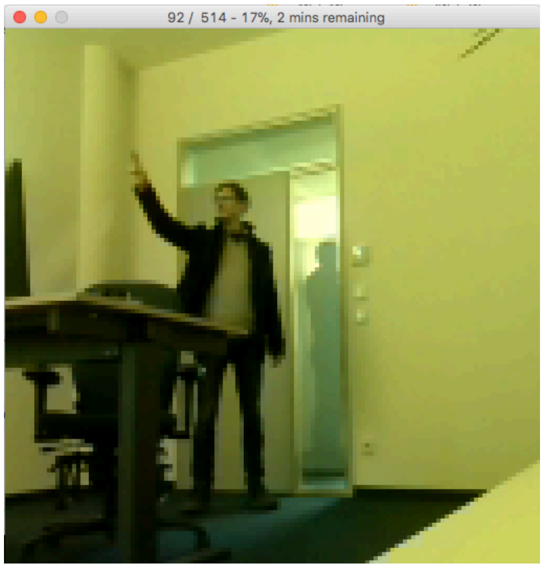
1. Record all the data in one go:

   ```
   python record.py day1 -1
   ```

   The -1 tells the **record.py** script to keep running until you kill it with ctrl-C.

2. After the images have been saved in day1/*.png, run the classify script:

   ```
   python classify.py day1
   ```

   
   92 / 514 - 17%, 2 mins remaining

3. Use these keyboard controls to classify the images:

   - 0-4: Classify the current image as '0' through '4'.
   - 9: Classify the next 10 images as '0' (useful as there are often a lot of "normal" images in a row).
   - Escape: Go back one image.
   - Space: Toggle viewer brightness to make human classification easier (does not modify the image file).
   - S: Move classified images to their new directories and quit.

   You can close the window without pressing S to exit without moving any files. This is not very user-friendly but provides just enough interface to perform the task.

   If you interrupted record.py with ctrl-c then the last image file is usually incomplete and cannot be loaded. In that case, classifying will show an error message in the title bar for that image and will not move it to a classification directory. You can delete this image manually after checking if it is unreadable.

4.2. What are we doing here?

Writing out each frame as a PNG file is a widely used technique which is compatible with the Keras tools for streaming data from disk during training. It allows easy inspection and curation of the training data, runs at 11 FPS, and generates approximately 700 MB of data per hour. This combination of good performance and easy access to the data makes this suitable for training a network from scratch with moderate amounts of data.

The standard way to store this data is in the following format:

```
DATA_DIR/CLASS/image.png
```

Here, `DATA_DIR` is a directory for this set of data (for example, the recordings from one particular day) and CLASS is the action class that the images represent.

For our demo video, 5 classes were used – one for each action.

You can record classes, individually. For example, by running:

```
python record.py day1/0 60
```

This will record 1 minute of images that should represent the *None* action (class 0). However, it is more efficient to record multiple actions together in a natural sequence, such as entering the room, pointing at the light, sitting down, getting up and then leaving the room. When doing this you need to label the images with their class after recording ends, and split them into separate directories, as described in the next section.

Advanced information
There are several other ways of collecting and storing your data, each with their own strengths and weaknesses. Here are some of these alternatives, for future reference:

- Run the MobileNet feature extractor during recording and only saving its output features.

  With a slightly modified feature extractor (an AvgPool layer at the end to reduce the output features to just 256 floats) this generates around 27 MB of data per hour of recorded video.

  This is very efficient and eliminates any concerns about memory usage. However, it reduces the frame rate to 7 FPS and as only the features are stored and not the images there is no way to examine the recorded images to understand problems that occur during training.

  This method forces you to stick with one feature extractor; to train a new neural network from scratch or try a different feature extractor you will have to recollect all the data.

- Use HDF5 to read and write files.

  The Hierarchal Data Format (HDF5) is designed to store and organize large amounts of data. Python's h5py module can be used to append frames to a file directly, rather than buffering them all in memory. This keeps access to the raw images whilst avoiding the need to store many images in memory during recording or training.

  This reduces the frame rate to 8 FPS on a Raspberry Pi 3 and generates around 8 GB of data per hour of recorded video. This is storage-intensive and the images are not particularly convenient to inspect while debugging training problems.

- Record video and post-process it to extract frames.

  This is the most efficient option and allows a high frame rate to be captured and stored. When recording very long sequences, we recommend considering this option.

  However, there are a few issues to be aware of. Firstly, recording compressed video does not perform as well if you are randomly fluctuating the white balance of the camera to add robustness to different lighting conditions. Secondly, lossy compression can cause artifacts in individual frames that do not appear in your "real" data while the network is being used. This can, for example, teach a network to identify objects based on the compression artifacts caused by their patterns. If possible, it is best to avoid such complexities.

5 Stream data from files during training

To create a validation data set, split off around 10% of the training data into a validation set. The validate_split.py helper script does this:

```
python validate_split.py day1
```

Keras has built-in support for streaming images from files in this directory structure through the ImageDataGenerator class and its flow_from_directory method. However, when training it is valuable to have a *validation* data set, which is taken from the same set of images as the training data but is set aside and used to estimate how well a network is generalizing to images that it has not seen during training.

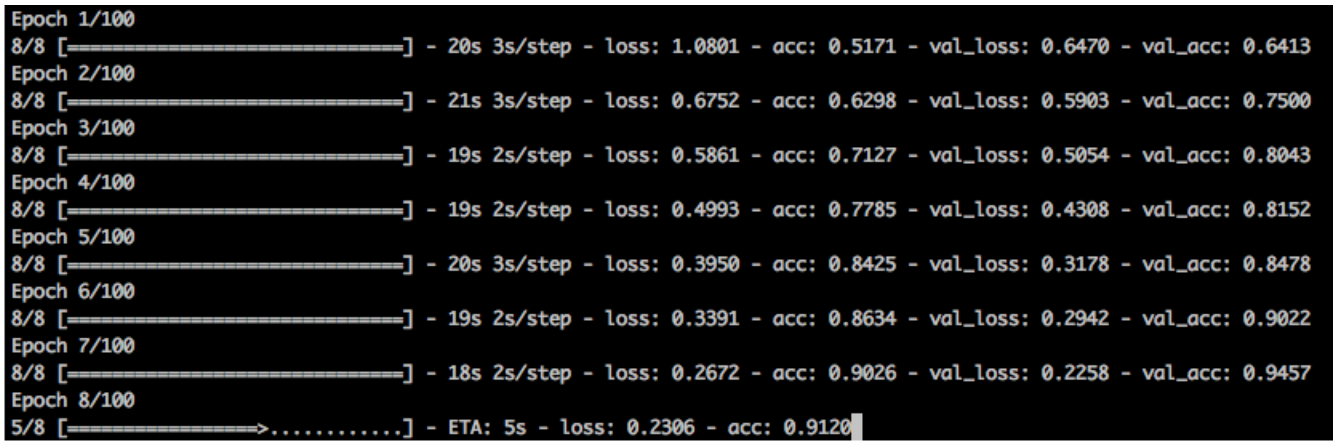
6 Train a neural network from scratch

Training a convolutional network is very compute-intensive and will take a long time on a Raspberry Pi 3. It will be quicker to copy the files to a laptop or desktop and run the train.py script there. To do this you will need to install TensorFlow on your laptop or desktop by following this guide.

To train a neural network from scratch with the LeNet-like model using your training data and validation data, use this command:

```
python train.py day1 val_day1
```

This shows regular progress updates while training the model, as shown here:



```
Epoch 1/100
8/8 [==============================] - 20s 3s/step - loss: 1.0801 - acc: 0.5171 - val_loss: 0.6470 - val_acc: 0.6413
Epoch 2/100
8/8 [==============================] - 21s 3s/step - loss: 0.6752 - acc: 0.6298 - val_loss: 0.5903 - val_acc: 0.7500
Epoch 3/100
8/8 [==============================] - 19s 2s/step - loss: 0.5861 - acc: 0.7127 - val_loss: 0.5054 - val_acc: 0.8043
Epoch 4/100
8/8 [==============================] - 19s 2s/step - loss: 0.4993 - acc: 0.7785 - val_loss: 0.4308 - val_acc: 0.8152
Epoch 5/100
8/8 [==============================] - 20s 3s/step - loss: 0.3950 - acc: 0.8425 - val_loss: 0.3178 - val_acc: 0.8478
Epoch 6/100
8/8 [==============================] - 19s 2s/step - loss: 0.3391 - acc: 0.8634 - val_loss: 0.2942 - val_acc: 0.9022
Epoch 7/100
8/8 [==============================] - 18s 2s/step - loss: 0.2672 - acc: 0.9026 - val_loss: 0.2258 - val_acc: 0.9457
Epoch 8/100
5/8 [==================>...........] - ETA: 5s - loss: 0.2306 - acc: 0.9120
```

Although the output suggests that the network will train for 100 epochs, this is the maximum value, and in practice it finishes earlier than this. The day1/model.h5 file is updated whenever a better result is achieved, so it is possible to leave training running, then copy the best model.h5 file that gets produced to the Raspberry Pi and try it out to decide whether it is already good enough for real-world use.

Note: A GPU will speed this up but is not necessary. With 2500 images, the models train in under an hour on a 2017 MacBook Pro.

6.1. What are we doing here?

Let's take a look at what's going on in train.py.

The simple LeNet architecture features blocks of convolution, activation, and max pooling followed by one or more dense layers. This architecture works well for a wide range of applications and is small enough to run at around 10 FPS on a Raspberry Pi 3.

The code in train.py sets up a simple convolutional network following this pattern:

```
model = Sequential()
model.add(Conv2D(32, (3, 3), input_shape=(128, 128, 3)))
model.add(Activation('elu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(32, (3, 3)))
model.add(Activation('elu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(64, (3, 3)))
model.add(Activation('elu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Flatten())
model.add(Dense(64))
model.add(Activation('elu'))
model.add(Dropout(0.5))
model.add(Dense(nb_classes))
model.add(Activation('softmax'))

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

You can increase or decrease the capability of the network by increasing and decreasing the channels (the first argument to each Conv2D call) and the size of the dense layer. In the code shown, these are set to 32, 32, 64, 64 respectively. To detect just one gesture (such as pointing at a light to turn it on or off) a network using 16, 16, 32, 16 trains and runs twice as fast with no loss in accuracy, so feel free to experiment with these values.

Once a good model has been found it can be instructive to come back and explore the effect of different activation functions such as relu and selu, the amount of dropout used, adding batch normalization or additional layers and so on. In general, this is not necessary as the above defaults already work well across multiple tasks.

The train.py script used in the previous tutorial loaded all the data into memory before training, which limited the number of images that could be used. This version uses Keras' ImageDataGenerator to stream the images from disk without loading them all at once:

```
train_datagen = ImageDataGenerator(
    rescale=1./255,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=False)

val_datagen = ImageDataGenerator(rescale=1. / 255)

train_generator = train_datagen.flow_from_directory(
    train_data_dir,
    target_size=(128, 128),
    batch_size=batch_size,
    class_mode='categorical')

val_generator = val_datagen.flow_from_directory(
    val_data_dir,
    target_size=(128, 128),
    batch_size=batch_size,
    class_mode='categorical')
```

This code also uses ImageDataGenerator's data augmentation to randomly shear and zoom each image in the training set by up to 20% each time it is seen by the neural network. This helps to make sure the neural network does not overfit to specific locations or sizes without having to move the camera between each recording.

When training a convolutional neural network from scratch instead of just fitting a classifier to features as in the previous tutorial, it helps to use a few extra tricks:

```
model.fit_generator(
    train_generator,
    steps_per_epoch=nb_train_samples // batch_size,
    epochs=100,
    validation_data=val_generator,
    validation_steps=10,
    callbacks=[ ModelCheckpoint('%s/model.h5' % train_data_dir, save_best_only=True),
                EarlyStopping(patience=10),
                ReduceLROnPlateau(factor=0.2, patience=5, verbose=1)])
```

The three callbacks shown here each help improve training and generalization performance:

- ModelCheckpoint: this ensures that the final model.h5 file saved is the one with the best score on the validation dataset, even if the model overfit the training data in subsequent epochs.
- EarlyStopping: this stops training after validation performance does not improve for more than 10 epochs to help prevent overfitting the validation dataset itself.
- ReduceLROnPlateau: this decreases the learning rate when training performance levels off and improves the overall performance of the model without having to fine-tune the learning-rate parameter.

You can explore the source code of train.py to see how these pieces fit together.

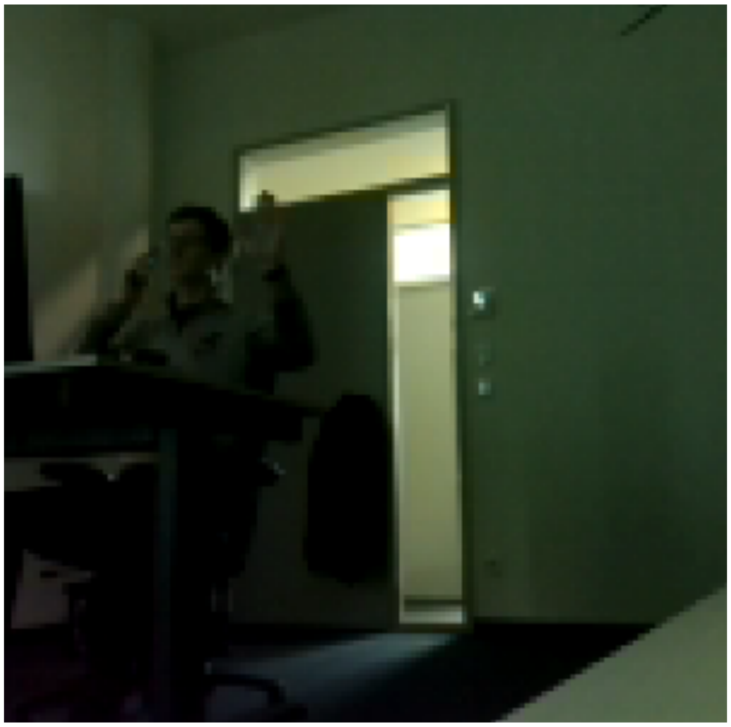
Advanced information

In Episode 1, we use a pre-trained MobileNet as a feature extractor, but one of the limitations of this approach is that most real-world gesture recognition situations do not look much like ImageNet, shown here:

(Image © Alec Radford, 2015. Used under an **MIT licence**)

This is a well-known problem when using models trained with ImageNet for transfer learning. The ImageNet data mostly consists of by well-lit, centered photographs with no noise or camera distortion. Because of this, networks trained on this data do not perform very well on images that do not share those characteristics.

A typical image used for gesture recognition in a dimly-lit office from a Raspberry Pi looks more like this:



From this image, can you tell that the subject has their left hand raised? Here, the hand and arm involve only 2% of the total pixels and have very low contrast compared to background parts of the image. This dissimilarity to ImageNet images suggests that training a neural network from scratch will be a better approach than trying to transfer ImageNet learning.

Training a convolutional neural network from scratch can be easy. There are many clever architectures that can be used to get the best possible performance on complex datasets, but to recognize a few gestures in a single location a simple architecture works fine.

7 Test the network on held out data

To evaluate whether a model will work in the real world it is useful to build a test dataset.

To do this, modify **record.py** and initialize the camera with **training_mode** set to **False**instead of **True**, like this:

```
    camera = Camera(training_mode=False)
    record(camera, name, seconds)
```

Now use it to record a single set of all the gestures you want to be recognized:

python record.py test1 -1

The -1 indicates that **record.py** should keep recording until you stop it by pressing ctrl-c on the keyboard, but you can also provide a fixed number of seconds if you prefer.

There are two important features of this test data:

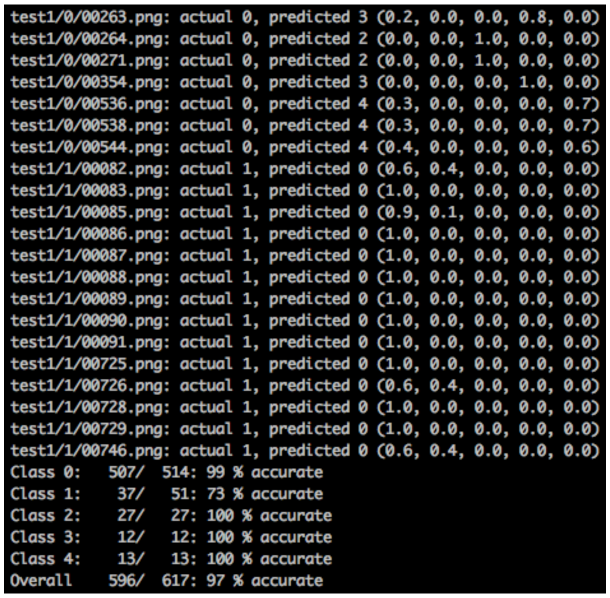
1. The training process of the network should never see it, not even as validation data.
2. It should be different from the training data in the same way that real-world usage might be different.

To test a network that should work whatever clothing you are wearing, record the test set wearing different clothes to the training set. Likewise, if you want the network to generalize to people that it has not seen before, make sure a previously unseen person is recorded in the test set.

Use the **test.py** script to load the model from the training directory and run each image in the test directory through it:

```
python test.py day1 test1
```

Incorrectly predicted images are listed along with their expected class and the probabilities the network predicted, allowing you to open the files and start to understand why the neural network did not recognize them. After testing on all images, the prediction accuracy for each class as well as overall is printed:



```
test1/0/00263.png: actual 0, predicted 3 (0.2, 0.0, 0.0, 0.8, 0.0)
test1/0/00264.png: actual 0, predicted 2 (0.0, 0.0, 1.0, 0.0, 0.0)
test1/0/00271.png: actual 0, predicted 2 (0.0, 0.0, 1.0, 0.0, 0.0)
test1/0/00354.png: actual 0, predicted 3 (0.0, 0.0, 0.0, 1.0, 0.0)
test1/0/00536.png: actual 0, predicted 4 (0.3, 0.0, 0.0, 0.0, 0.7)
test1/0/00538.png: actual 0, predicted 4 (0.3, 0.0, 0.0, 0.0, 0.7)
test1/0/00544.png: actual 0, predicted 4 (0.4, 0.0, 0.0, 0.0, 0.6)
test1/1/00082.png: actual 1, predicted 0 (0.6, 0.4, 0.0, 0.0, 0.0)
test1/1/00083.png: actual 1, predicted 0 (1.0, 0.0, 0.0, 0.0, 0.0)
test1/1/00085.png: actual 1, predicted 0 (0.9, 0.1, 0.0, 0.0, 0.0)
test1/1/00086.png: actual 1, predicted 0 (1.0, 0.0, 0.0, 0.0, 0.0)
test1/1/00087.png: actual 1, predicted 0 (1.0, 0.0, 0.0, 0.0, 0.0)
test1/1/00088.png: actual 1, predicted 0 (1.0, 0.0, 0.0, 0.0, 0.0)
test1/1/00089.png: actual 1, predicted 0 (1.0, 0.0, 0.0, 0.0, 0.0)
test1/1/00090.png: actual 1, predicted 0 (1.0, 0.0, 0.0, 0.0, 0.0)
test1/1/00091.png: actual 1, predicted 0 (1.0, 0.0, 0.0, 0.0, 0.0)
test1/1/00725.png: actual 1, predicted 0 (1.0, 0.0, 0.0, 0.0, 0.0)
test1/1/00726.png: actual 1, predicted 0 (0.6, 0.4, 0.0, 0.0, 0.0)
test1/1/00728.png: actual 1, predicted 0 (1.0, 0.0, 0.0, 0.0, 0.0)
test1/1/00729.png: actual 1, predicted 0 (1.0, 0.0, 0.0, 0.0, 0.0)
test1/1/00746.png: actual 1, predicted 0 (0.6, 0.4, 0.0, 0.0, 0.0)
Class 0:   507/  514: 99 % accurate
Class 1:    37/   51: 73 % accurate
Class 2:    27/   27: 100 % accurate
Class 3:    12/   12: 100 % accurate
Class 4:    13/   13: 100 % accurate
Overall    596/  617: 97 % accurate
```

Note: Some terminals, such as iTerm on the Mac, will interpret the filenames as links, allowing you to click on them to open the image directly. For the model we tested, most of the errors were in predicting class 1 (opening and closing the door). When manually inspecting these images it is clear in every case that the door is ever so slightly open and in fact, the classification label in the test set is arguably incorrect.

8 Improve generalization by recording more data

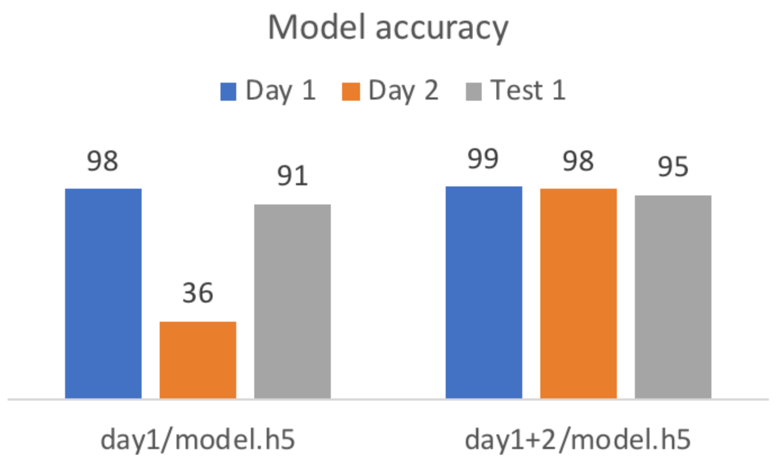
Your first model will probably perform much better against the training data than the test data. It has not yet been taught to ignore features that should be ignored, such as clothing or daylight. Generalization is improved by adding more varied data.

The simplest way to approach this is to keep on adding more data until doing so produces no further improvement.

Instead of spending time trying to improve the test performance directly, you can just record another set of data in different conditions. Here, this second data set is called day2. To train a model on the combined data sets from day 1 and day 2, simply merge their directory structures. The merge.py helper script does this automatically:

```
python merge.py day1+2 day1 day2
```

The above creates a new directory called day1+2 and copies all the images in day1/*/*.png and day2/*/*.png to it, maintaining the classification into subdirectories. Now it is straightforward to train a new model on the combined data set using the train.py script and compare its performance on all three data sets (day1, day2 and test1) with the previous model using the test.py script. For example:



Adding a second day of data with different lighting and clothing not only improved the performance on the unseen test set, it also improved the performance of the day1 data. Notice that although the model trained on day1 performed reasonably on the test data, it got only 36% of its predictions correct on the day2 data.

Day1 and day2 both use the randomized white balance, making classification harder, so this is always going to be worse than the test set, but this very low score suggests training only on day1 was not enough to generalize to new situations.

Is 95% accuracy good enough? For some applications, it might be, but mispredicting 1 frame in 20 is once every two seconds at 10 FPS. Depending on when these occur, this could be very problematic. It is easy to keep adding more data to help improve the accuracy. Applying the same process again on the third day looks like this:

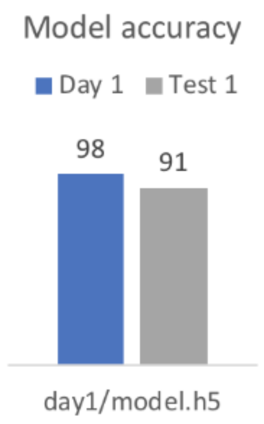
```
python record.py day3
python classify.py day3
python merge.py day1+2+3 day1 day2 day3
python validate_split.py day1+2+3
python train.py day1+2+3 val_day1+2+3
python test.py day1+2+3 day1
python test.py day1+2+3 day2
python test.py day1+2+3 day3
python test.py day1+2+3 test1
```

Advanced information

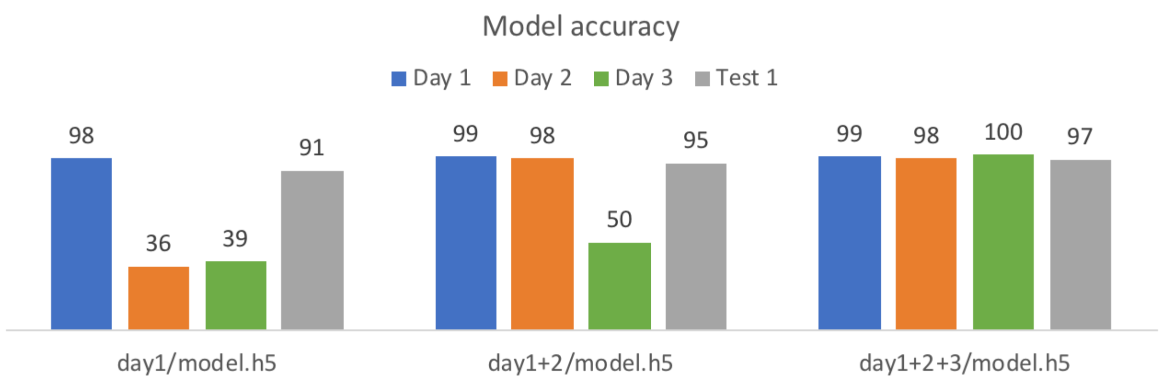
To prepare the demo system shown in the video at the start of this guide, 3-4 minutes of video was recorded every day for three days. Half of this was filmed in dim lighting and half filmed in bright lighting. Each one of the five actions was repeated with several variations and the results were classified and saved.

After training a ConvNet from scratch on the day1 dataset, it achieved an accuracy of 98% on its training data and 98% on the validation set. The high validation accuracy shows that the model has not just memorized the input data, but as the validation data was drawn from the same day it does not tell us whether the model will generalize to different clothes, ambient daylight and so on.

The test set performance is good but not great - making a mistake 9% of the time leads to many misinterpreted gestures:



As more data is added from subsequent days, the performance improves across on every dataset:



Here we can see that although neither day1/model.h5 nor day1+2/model.h5 were trained on the day3 data, having seen two days worth of data meant that day1+2/model.h5 handled it significantly better. Adding the 3rd day of data almost halved the error on the test set and examining the remaining errors shows many pictures that were arguably mislabelled by hand in the classification process.

For our demo video, this was sufficient for reliable predictions but each use case will vary. To get the best performance, keep on adding extra datasets until there is no extra benefit from training on them.

This process ends in a single model file, such as day1+2+3/model.h5, which produces good performance on all the data seen so far as well as the test set. All that remains now is to deploy it.

9 Deploy the network with actions

Here, you are going to deploy the network with actions for using gestures to trigger audio file playback and to control lighting.

First, copy the trained model.h5 file back over to the Raspberry Pi 3 from the PC where you ran the train.py script.

The run.py script will execute the model using the following code:

```
def enter(self):
    stderr.write('\nEnter\n')
    self.light.on()
    mixer.music.load('audio/good-evening.wav')
    mixer.music.set_volume(1.0)
    mixer.music.play()
    system('ssh zero ./unlock &')

def toggle_light(self):
    if self.light.value:
        stderr.write('\nLight off\n')
        self.light.off()
    else:
        stderr.write('\nLight on\n')
        self.light.on()

def play_music(self):
    stderr.write('\nMusic play\n')
    mixer.music.load('audio/epic.mp3')
    mixer.music.set_volume(0.5)
    mixer.music.play()

def dim_music(self):
    stderr.write('\nDim music\n')
    mixer.music.set_volume(0.1)

def leave(self):
    stderr.write('\nLeave\n')
    mixer.music.fadeout(1)
    mixer.music.load('audio/good-night.wav')
    mixer.music.set_volume(1.0)
    mixer.music.play()
    self.light.off()
    system('ssh zero ./lock &')
    sleep(5)
    mixer.music.load('audio/alarm-active.wav')
    mixer.music.play()
```

Note: This is slightly simpler than in Episode 1 as we do not need to convert the pixels into features with MobileNet. Instead, the ConvNet directly converts pixels into predictions.

You can run this with the following command:

```
python run.py model.h5
```

A continuous update of the predicted class is printed to stderr, but as yet no actions are taken. This is a useful baseline for your own projects.

For the demo video, the file story.py was created. This uses the same code but performs different actions based on the predicted class.

To recap, the list of classes used in this demo are:

Class	Action
0	None - moving around, sitting, putting on and taking off coat.
1	Door - coming in or leaving through the door
2	Light - pointing at the main light to turn it on/off
3	Music - holding both hands up to start the music playing
4	Stop - holding up one hand to dim the volume of the music

There is only one action for Class 1 (Door) as the network was not trained to distinguish between somebody coming into the room and leaving it. This would have increased the complexity of recording and handling the data. Instead, the Room class tracks the state of the room, assuming that there is only ever one person who uses it. So the first time the door opens it must be somebody coming in the room and the next time it must be somebody going out of the room.

This is an oversimplification of the real world that would only work for a demo. A more robust alternative is to train the same, or a separate, network to independently detect whether the room is empty or not. However, this is outside the scope of this guide in order to keep the training section here as straightforward as possible.

One of the benefits of machine learning directly on devices, known as "on the edge", is how easy it is to combine it with regular programming to build more complex behaviors. The simple state machine controlling the demo in story.py looks like this:

```python
def update(self, action):
    if action == NONE:
        self.ready = True

    if not self.ready:
        return

    if action != NONE:
        self.ready = False

    if action == DOOR:
        if self.state == 'empty':
            self.enter()
            self.state = 'occupied'
        elif self.state == 'occupied':
            self.leave()
            self.state = 'empty'
    elif action == LIGHT:
        self.toggle_light()
    elif action == MUSIC:
        self.play_music()
    elif action == STOP:
        self.dim_music()
```

Part of the power of training neural networks is that they provide a great API to the real world. In this case, the python code can tell whether someone is opening the door or pointing at a light by checking a simple variable!

The only complexity above is the use of self.ready, where the state machine is ready for a new action once the previous action has finished and the state returns to 0. This is a simple way to move from predicting states (is someone pointing at the light?) to actions (toggle the light when someone points at it, but do not toggle it on and off continuously while they are pointing at it).

The Raspberry Pi sits at a great intersection of the physical and information worlds. It can send email and push notifications, it can also turn home electrical appliances and lights on and off, it can play and record video and sound. Here, a few simple actions are used:

- Playing sounds and music.
- Locking and unlocking a remote computer.
- Turning on and off a standing lamp.

The entire code for these actions is:

```
def enter(self):
    stderr.write('\nEnter\n')
    self.light.on()
    mixer.music.load('audio/good-evening.wav')
    mixer.music.set_volume(1.0)
    mixer.music.play()
    system('ssh zero ./unlock &')

def toggle_light(self):
    if self.light.value:
        stderr.write('\nLight off\n')
        self.light.off()
    else:
        stderr.write('\nLight on\n')
        self.light.on()

def play_music(self):
    stderr.write('\nMusic play\n')
    mixer.music.load('audio/epic.mp3')
    mixer.music.set_volume(0.5)
    mixer.music.play()

def dim_music(self):
    stderr.write('\nDim music\n')
    mixer.music.set_volume(0.1)

def leave(self):
    stderr.write('\nLeave\n')
    mixer.music.fadeout(1)
    mixer.music.load('audio/good-night.wav')
    mixer.music.set_volume(1.0)
    mixer.music.play()
    self.light.off()
    system('ssh zero ./lock &')
    sleep(5)
    mixer.music.load('audio/alarm-active.wav')
    mixer.music.play()
```

You can write any code you want here to modify this for your own system. At a minimum, you will need to provide an audio/epic.mp3 file, as the one used in the demo is not redistributed in the GitHub repository. You will also need to change or remove the "ssh zero ..." commands to lock and unlock the screen - in the demo video, the screen was driven by a Raspberry Pi Zero, with passwordless SSH configured to allow the Pi 3 running the network to execute a command to lock or unlock the screen.

The self.light.on() and self.light.off() commands are provided by the Energenie class in the gpiozero python module. The demo was run with an Energenie Pi-Mote and socket, which uses a radio link between the Raspberry Pi and a socket to allow it to be turned on and off via software.

10 Next steps

At this stage you can create your own home AI using trained neural networks, python, and the Raspberry Pi. The helper scripts are all available in our GitHub repository.

To learn more about how convolutional networks work and how to build them in Keras, see Andrej Karpathy's excellent visual explanation and Francois Chollet's Keras introduction.

The next, and final, tutorial in this series will combine some of these techniques into a demo that allows people to train and use neural networks with no programming required, but as this tutorial shows: combining neural networks with a little programming expertise opens up a fascinating world of opportunities to explore!

If you are interested in using low-power, always-on sensors then take a look at our work running neural networks on low-power Cortex-M CPUs!