CS 5600/6600: F21: Intelligent Systems
Project 1: Image and Audio Classification with ANNs and ConvNets with TFLearn and Keras

Vladimir Kulyukin
Department of Computer Science
Utah State University

September 23, 2021

Learning Objectives

1. Artificial Neural Networks (ANNs)
2. Convolutional Networks (CNs)
3. Image and Audio Classification
4. TFLearn, Keras

Introduction: On to Open Science

Let me say a few words about this project to give you a broader picture of the intellectual background behind it. Back in 2013, I started seeing many publications in the apiary science literature that I follow as a beekeeper about failing honeybee colonies all over the world. The high rates of colony loss not only threaten to disrupt our food supply, which is quite a threat in and of itself, but also may tell us disturbing stories about the environment that we are unable to hear yet. For example, for U.S. beekeepers, the average colony loss varies from 25% to 50% per year. The cost of equipment, bee packages, maintenance, and transportation is so high that profit margins for beekeepers are very small. Pursuant to my principal research criterion that my systems not only advance science, but also affect social change, I asked myself what I could do as a computer scientist and a beekeeper to improve the health of honeybee colonies.

In 2013–2014, after taking a deep dive into the apiary science literature, I saw an emerging consensus among some entomologists, computer scientists, and engineers that electronic beehive monitoring (EBM) can help extract critical information on colony behavior and phenology without invasive inspections and significant transportation costs.

In 2014, as good luck would have it, I also came across Michael Nielsen's book *Reinventing Discovery: The New Era of Networked Science* (Princeton University Press, 2011). I learned from this book about *Polymath*, a crowdsourced mathematics project, where spontaneous virtual communities all over the world collaborate to solve previously unsolved problems, and about *Galaxy Zoo*, a crowdsourced astronomy project, where over 250,000 amateur astronomers work together to understand the structure of the Universe and discover new galaxy types. In my subsequent research, I learned about *Bee Informed*, an apiary science network, where thousands of beekeepers collaborate to improve beekeeping practices and make apiary science discoveries.

I find these projects remarkable signs of forward thinking. These give us road maps of how researchers, amateur and professional alike, work together to advance the cause of Open Science,

because traditional (aka closed/institutionalized science) science is experiencing two rapidly growing crises: 1) crisis of replicability/reproducibility and 2) crisis of peer review.

No, I'm not kidding about these crises in traditional science. In 2005, Dr. John Ioannidis, a reseacher, also a physician and now a prolific writer, at the Stanford Prevention Research Center, published his landmark article "Why Most Published Research Findings Are False" [1], where he presented a computational model that showed (conclusively to many subsequent reviewers) that "for many current scientific fields, claimed research findings may often be simply accurate measures of the prevailing bias." This paper became an icon in the open science community worldwide, because it was one of the first groundbreaking publications to succinctly describe the replication/reproducibility and review crises in science.

To many researchers, including my humble self, Dr. Ioannidis' article was the proverbial canary in the coal mine, because it shows that, although reproducibility is a cornerstone of the scientific method, a large percentage of scientific results published in "top-tier" academic journals are irreproducible/irreplicable (i.e., useless at best and misleading at worst).

In 2011, Nature published a paper [2] showing that researchers could reproduce between 20 and 25% of 67 published preclinical drug studies. In 2012, Nature published another paper [3] showing that researchers could reproduce only 6 out of a total of 53 "landmark" cancer studies (a reproducibility rate of 11%). A study published in 2012 [4] found that fraud accounts for 43% of scientific paper retractions from "top-tier" academic journals and demonstrates a 1000% increase in scientific fraud (reported, of course) since 1975. I note in passing that there's now a site www.retractionwatch.org that monitors publication retractions and fraud. A recent top post on that cite was "Publisher offers cash for citations." Real funny if it weren't so sad.

In May 2016, Nature published the results of a survey of over 1,500 scientists ("1,500 Scientists Lift the Lid on Reproducibility"), which found that 70% of them failed to reproduce published experimental results [5]. The study covered not only medical researchers but also researchers from physics, chemistry, earth and environmental sciences, and other assorted areas.

My grad students and I can relate to the observations in [5]. When we asked several IEEE/ACM authors to send us their source code/hardware design/data to replicate and build upon their results, we received no answers like "I don't think it exists any more" or "we're considering applying for a patent and cannot give it to you at this time." The story is the same with "proprietary datasets" hoarded for a competitive advantage.

Some smarter institutionalists heed the harbinger, and have started to create public crowdfunding sites to embrace the change. My long-term bet, however, is that they'll fail or, at best, show the same lackluster performance as the www.citizenscience.gov initiative, which appears to have issued its last report back in 2017-2018. Hierarchies cannot contain spontaneous, informal, and highly motivated networks. In the long run, the bazaar always outsmarts the cathedral [6]. A caveat: when reading the latter book (and it is worth reading for any Linux hacker!), one shouldn't confuse Open Science with Open Source. Open Science is not just about software – it requires equipment and, in the context of this project, honeybees and land resources.

In 2017, I created and ran my first science crowdfunding project *BeePi: A Multisensor Electronic Beehive Monitor* on Kickstarter to crowdfund some basic hardware needs for my electronic beehive monitoring (EBM) research. Since this was my first crowdfunder, my target goal was a modest $1,000. I was genuinely amazed that within 60 days 61 backers pledged $2,940 to bring my project to life. In 2019, I ran another science crowdfunding project on Kickstarter *BeePi: Honeybees Meet AI: Stage 2* with the target goal of $5,000. I was again pleasantly surprised that 2 months later 59

backers pledged $5,753 to bring my project to life.

The wonderful datasets (actually, the smaller versions thereof) described in the next section we'll be working with in this project come courtesy of our generous Kickstarter backers whose donations helped us to purchase the hardware, hive woodenware, and bee packages, and capture the data. My point is that research can (and should!) be crowdfunded and that science crowdfunding is possible. Random acts of kindness and generosity go a long way.

Before we dive into the datasets, I'd like to thank, from the bottom of my heart, all my wonderful graduate and undergraduate students who've been working shoulder to shoulder with me, mostly pro bono, on acquiring, curating, and experimenting with these datasets: Myles Putnam (this mega-bright guy invented the name BeePi), Prakhar Amlathe, Jit Mukherjee, Kristoffer Price, Anastasiia Tkachenko, Daniel Hornberger, Nikhil Ganta, Matthew Lister, Aditya Bhouraskar, Laasya Alavala, Chelsi Gupta, Astha Tiwari, Keval Shah, Matthew Ward, Sai Kiran Reka, and Sarat Kiran Andhavarapu. I'd like to thank Trevor Landeen, Ashwani Chahal, Felipe Queiroz, Dinis Quelhas, Tharun Tej Tammineni, and Tanwir Zaman for their generous Kickstarter contributions and express my gratitude to Gregg Lind, who backed both fundraisers and donated hardware to the BeePi project. I'm also grateful to Richard Waggstaff, Craig Huntzinger, and Richard Mueller for letting me use their private property in northern Utah for longitudinal EBM tests.

Datasets

I uploaded the smaller versions of the datasets in Canvas (See CS5600/6600: F21: Project 1: TFLearn Datasets and CS5600/6600: F21: Project 1: Keras Datasets). These datasets were obtained from the data captured by the BeePi monitors deployed on live beehives in Logan and North Logan in 2017 – 2018.

There are 6 image datasets in the zip archive for this project: `BEE1`, `BEE1_gray`, `BEE2_1S`, `BEE2_1S_gray`, `BEE_4`, `BEE_4_gray`. There are 3 audio datasets: `BUZZ1`, `BUZZ2`, `BUZZ3`. The suffix `_gray` in an image dataset's name means that the dataset is a grayscale version of the corresponding color dataset. For example, `BEE1_gray` is the grayscale version of `BEE1`. The 3 grayscale datasets (i.e., `BEE1_gray`, `BEE2_1S_gray`, `BEE_4_gray`) will be used in the training and testing of ANNs. The other 3 datasets (i.e., `BEE1`, `BEE2_1S`, `BEE_4`) will be used in the training and testing of ConvNets. The audio datasets can be used in the training and testing of both ANNs and ConvNets.

Each dataset consists of 6 pickle files: `train_X.pck`, `train_Y.pck`, `test_X.pck`, `test_Y.pck`, `valid_X.pck`, `valid_Y.pck`. The `_X` files include examples, the `_Y` files contain the corresponding targets (i.e., ground truth). The `train_` and `test_` files can be used in training and testing. The `valid_` files can be used in validation.

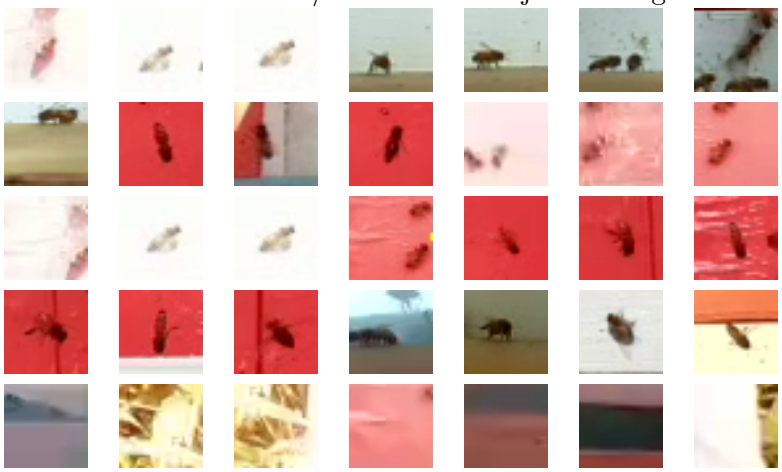
The Project 1 zip contains 3 audio files (`buzz.wav`, `cricket.wav`, `noise.wav`) to give you examples of audio files out of which the numpy arrays of `BUZZ1`, `BUZZ2`, and `BUZZ3` were computed and normalized.

TFLearn and Keras Versions

To get this project to run, I had to make two virtual envronments (separate work spaces) on my Linux. For TFLearn, I used TFLearn 0.3.2 and Python 3.6.8. To install TFLearn, I followed the instructions at tflearn.org/installation/.

I couldn't get Keras to work with Python 3.6.8. After sinking 5+ hours into it, I gave up and created

Table 1: CS5600/6600: F21: Project 1 Images



another virtual environment, installed Python 3.6.9 into it (as some technical blogs recommended), and followed the steps below. I'm skipping the installation of third-party libraries, updates, and dependencies (another 2+ hours – deep sigh!). I also read https://keras.io and some related links.

1. install tensorflow on python 3.6.9 with `pip3 install tensorflow==2.4.1`;
2. install scikit-image on python 3.6.9 with `pip3 install scikit-image`;
3. install sklearn on python 3.6.9 with `pip3 install sklearn`.

Loading TFLearn Datasets

The datasets for the TFLearn part of this project contains the following datasets: BEE1, BEE1_gray, BEE2_1S, BEE2_1S_gray, BEE4, BEE4_gray, BUZZ1, BUZZ2, BUZZ3. The datasets BEE1, BEE1_gray, BEE2_1S, BEE2_1S_gray, BEE4, BEE4_gray are video datas; the datasets BUZZ1, BUZZ2, BUZZ3 are audio. The datasets contain pickle (i.e., `.pck`) files and contain pre-processed images and audio files that can be loaded into Python directly. Note that BEE1_gray and BEE2_1S_gray contain grayscale images (i.e., 64x64x1) that we'll use to train ANNs. ConvNets will be trained on color images (i.e., 64x64x3).

The project zip contains `tfl_audio_anns.py`, `tfl_audio_convnets.py`, `tfl_image_anns.py`, and `tfl_image_convnets.py` that show you how to load the processed datasets, create ANNs and ConvNets with tflearn.

Let's run `tfl_image_anns.py`.

```
>>> from tfl_image_anns import *
loading datasets from BEE1_gray/...
...
datasets from BEE4_gray/ loaded...
```

Let's check the shape of the `BEE1_gray` training examples and targets.

```
>>> BEE1_gray_train_X.shape
(38139, 64, 64, 1)
>>> BEE1_gray_train_Y.shape
(38139, 2)
```

The above output shows that the training set consists of 38,139 64x64x1 numpy arrays. The last 1 means that the image has 1 channel (i.e., it is grayscale). The corresponding targets consists of 38,139 2-element numpy arrays. Let's print the first 10 targets.

```
>>> BEE1_gray_train_Y[:10]
array([[0., 1.],
       [0., 1.],
       [0., 1.],
       [0., 1.],
       [0., 1.],
       [0., 1.],
       [1., 0.],
       [1., 0.],
       [0., 1.],
       [1., 0.]])
```

If a target is `[1., 0.]`, the corresponding example is classified as `BEE`. When a target is `[0., 1.]`, it means that the corresponding example is classified as `NO_BEE`. For example, since `BEE1_gray_train_Y[0]` is `[0., 1.]`, `BEE1_gray_train_X[0]` is classified as `NO_BEE`.

Let's play with the audio datasets and load `tfl_audio_anns.py`. I'll skip some output lines below.

```
>>> from tfl_audio_anns import *
loading datasets from BUZZ1/...
datasets from BUZZ1/ loaded...
loading datasets from BUZZ2/...
datasets from BUZZ2/ loaded...
loading datasets from BUZZ3/...
datasets from BUZZ3/ loaded...
```

Let's explore `BUZZ1`. The other two datasets (`BUZZ2` and `BUZZ3`) are similar.

```
>>> BUZZ1_train_X.shape
(7000, 4000, 1, 1)
>>> BUZZ1_train_Y.shape
(7000, 3)
>>> len(BUZZ1_train_X[0])
4000
>>> BUZZ1_train_Y[0]
array([1., 0., 0.])
>>> BUZZ1_train_Y[:10]
array([[1., 0., 0.],
       [1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.],
       [0., 1., 0.],
       [0., 0., 1.],
       [0., 1., 0.],
       [0., 1., 0.],
       [0., 0., 1.],
       [1., 0., 0.]])
```

The above interaction shows that each audio example is 4000x1 numpy array. It's 4,000 amplitude readings. Each target is a 3-element numpy array (e.g., `[1., 0., 0.]`). This is a 3-way classification – `BEE`, `CRICKET`, and `NOISE`. When a target is `[1., 0., 0.]`, the corresponding example is classified as `BEE`; when a target is `[0., 1., 0.]`, the corresponding example is classified as `CRICKET`; and when a target is `[0., 0., 1.]`, the corresponding example is classified as `NOISE`. Some of you asked me which tools we used to process the audio files. In case you're interested, I used `scipy.io.wavfile` and normalized the amplitudes to be between 0 and 1.

Training, Testing, and Validating ANN Image Models with TFLearn

The function `make_image_ann_model()` in `tfl_image_anns.py` shows you an example of defining an ANN with `tflearn`. An ANN is just a set of fully connected layers. The network has an input layer that takes 64x64x1 inputs (i.e., grayscale images), the input layer is fully connected to `fc_layer_1` (the hidden layer) of 128 ReLU neurons. The hidden layer is connected to `fc_layer_2` (the output layer) of 2 softmax neurons. The call to the `regression` function defines the learning rate of the network, the stochastic gradient descent as the weight optimization function, and the categorical cross entropy as the loss function (i.e., the cost function).

```
def make_image_ann_model():
    input_layer = input_data(shape=[None, 64, 64, 1])
    fc_layer_1 = fully_connected(input_layer, 128,
                                 activation='relu',
                                 name='fc_layer_1')
    fc_layer_2 = fully_connected(fc_layer_1, 2,
                                 activation='softmax',
                                 name='fc_layer_2')
    network = regression(fc_layer_2, optimizer='sgd',
                         loss='categorical_crossentropy',
                         learning_rate=0.1)
    model = tflearn.DNN(network)
    return model
```

Here's how we can make an ANN model.

```
>>> from tfl_image_anns import *
>>> img_ann = make_image_ann_model()
>>> img_ann
<tflearn.models.dnn.DNN object at 0x7faa39e90908>
```

Once we have a model, we can train it. The function `train_tfl_image_ann_model()` gives you an example of how to do it. Your output should look as follows.

```
>>> train_tfl_image_ann_model(img_ann,
                              BEE1_gray_train_X, BEE1_gray_train_Y,
                              BEE1_gray_test_X, BEE1_gray_test_Y)
---------------------------------
Run id: image_ann_model
Log directory: /tmp/tflearn_logs/
---------------------------------
```

```
Training samples: 38139
Validation samples: 12724
--
Training Step: 3814  | total loss: 0.66422 | time: 5.367s
| SGD | epoch: 001 | loss: 0.66422 - acc: 0.5912 | val_loss: 0.66742 -
                                                    val_acc: 0.5386 -- iter: 38139/38139
--
Training Step: 7628  | total loss: 0.65895 | time: 5.206s
| SGD | epoch: 002 | loss: 0.65895 - acc: 0.5784 | val_loss: 0.66352 -
                                                    val_acc: 0.5549 -- iter: 38139/38139
--
```

Our loss is pretty high and accuracy is low, but we've trained this ANN only for 2 epochs. No surprises here. Once we have a trained model, we need to do 2 things with it: validate it on a dataset that was not used in training/testing and, if it's good enough, persist. The above model is not that great, but I'll perist it anyway to show you how to do id. Let's validate it first.

```
>>> validate_tfl_image_ann_model(img_ann, BEE1_gray_valid_X, BEE1_gray_valid_Y)
0.5702947845804989
```

Now we can persist it in an appropriate `tfl` file.

```
>>> img_ann.save('/home/vladimir/teaching/AI/project_01/models/img_ann.tfl')
```

Check the directory where you persist your nets to make sure that everything is persisted. You should see files like `am_buzz1.tfl.index`, etc. Later on, if we want to use a persisted model or train it some more, we need to load it first and then use/train it.

```
>>> iam = load_image_ann_model('/home/vladimir/teaching/AI/project_01/models/img_ann.tfl')
>>> test_tfl_image_ann_model(iam, BEE2_1S_gray_valid_X, BEE2_1S_gray_valid_Y)
0.7147026632615834
```

Training, Testing, and Validating ConvNet Image Models with TFLearn

Let's construct, train, test, and validate a TFLearn ConvNet on images. The function `make_image_convnet_model()` in `tfl_image_convnets.py` shows you an example of defining an ANN with `tflearn`. This ConvNet contains a convolution layer followed by a max-pool layer and a fully connected hidden layer. The output layer has 2 output softmax neurons. Note that the shape of the input is 64x64x3, because these arrays were created from color (i.e., 3 channel) images.

```
def make_image_convnet_model():
    input_layer = input_data(shape=[None, 64, 64, 3])
    conv_layer_1 = conv_2d(input_layer,
                           nb_filter=8,
                           filter_size=3,
                           activation='relu',
                           name='conv_layer_1')
    pool_layer_1 = max_pool_2d(conv_layer_1, 2, name='pool_layer_1')
    fc_layer_1 = fully_connected(pool_layer_1, 128,
                                 activation='relu',
```

```
                                name='fc_layer_1')
    fc_layer_2 = fully_connected(fc_layer_1, 2,
                                activation='softmax',
                                name='fc_layer_2')
    network = regression(fc_layer_2, optimizer='sgd',
                         loss='categorical_crossentropy',
                         learning_rate=0.1)
    model = tflearn.DNN(network)
    return model
```

Let's make a ConvNet model.

```
>>> from tfl_image_convnets import *
...
datasets from BEE4/ loaded...
>>> img_cn = make_image_convnet_model()
>>> img_cn
<tflearn.models.dnn.DNN object at 0x7f515df5c5f8>
```

Let's train this ConvNet. The function `train_tfl_image_convnet_model()` gives you an example of how to do it. Your output should look as follows.

```
>>> train_tfl_image_convnet_model(img_cn, BEE1_train_X, BEE1_train_Y,
                                  BEE1_test_X, BEE1_test_Y)
---------------------------------
Run id: image_cn_model
Log directory: /tmp/tflearn_logs/
---------------------------------
Training samples: 38139
Validation samples: 12724
--
Training Step: 3814  | total loss: 0.13714 | time: 19.153s
| SGD | epoch: 001 | loss: 0.13714 - acc: 0.9503 | val_loss: 0.13552 -
                                                    val_acc: 0.9477 -- iter: 38139/38139
--
Training Step: 7628  | total loss: 0.13324 | time: 19.037s
| SGD | epoch: 002 | loss: 0.13324 - acc: 0.9591 | val_loss: 0.13543 -
                                                    val_acc: 0.9499 -- iter: 38139/38139
--
```

Our loss could be smaller and accuracy is 0.95. We need to train some more. Let's validate the model on a different dataset that was not used in training/testing and persist it.

```
>>> validate_tfl_image_convnet_model(img_cn, BEE1_valid_X, BEE1_valid_Y)
0.9316893424036281
>>> validate_tfl_image_convnet_model(img_cn, BEE2_1S_valid_X, BEE2_1S_valid_Y)
0.566216709230208
>>> validate_tfl_image_convnet_model(img_cn, BEE4_valid_X, BEE4_valid_Y)
0.5726284584980237
>>> img_cn.save('/home/vladimir/teaching/AI/project_01/models/img_cn.tfl')
```

We can now load the model into Python and use/train it. Here' how.

```
>>> cm = load_image_convnet_model('/home/vladimir/teaching/AI/project_01/models/img_cn.tfl')
>>> cm
<tflearn.models.dnn.DNN object at 0x7f2b228edda0>
>>> validate_tfl_image_convnet_model(cm, BEE1_valid_X, BEE1_valid_Y)
0.9316893424036281
>>> validate_tfl_image_convnet_model(cm, BEE2_1S_valid_X, BEE2_1S_valid_Y)
0.566216709230208
>>> validate_tfl_image_convnet_model(cm, BEE4_valid_X, BEE4_valid_Y)
0.5726284584980237
```

Training, Testing, and Validating ANN Audio Models

The file `tfl_audio_anns.py` contains examples of how to construct and train, test, and validate ANNs on the 3 audio datasets, persist them, and load them into Python. The interaction below shows you how to do these steps. I won't comment the steps, because they're very similar to the steps discussed in the previous two sections.

```
>>> from tfl_audio_anns import *
>>> aud_ann = make_audio_ann_model()
>>> train_tfl_audio_ann_model(aud_ann, BUZZ1_train_X, BUZZ1_train_Y,
                              BUZZ1_test_X, BUZZ1_test_Y)
---------------------------------
Run id: audio_ann_model
Log directory: /tmp/tflearn_logs/
---------------------------------
Training samples: 7000
Validation samples: 2110
--
Training Step: 700  | total loss: 0.93983 | time: 1.877s
| SGD | epoch: 001 | loss: 0.93983 - acc: 0.5913 | val_loss: 0.97629 -
                                                    val_acc: 0.6526 -- iter: 7000/7000
--
Training Step: 1400  | total loss: 0.75596 | time: 1.890s
| SGD | epoch: 002 | loss: 0.75596 - acc: 0.6911 | val_loss: 0.91760 -
                                                    val_acc: 0.6991 -- iter: 7000/7000
--
>>> validate_tfl_audio_ann_model(aud_ann, BUZZ1_valid_X, BUZZ1_valid_Y)
0.45652173913043476
>>> aud_ann.save('/home/vladimir/teaching/AI/project_01/models/aud_ann.tfl')
>>> am = load_audio_ann_model('/home/vladimir/teaching/AI/project_01/models/aud_ann.tfl')
<tflearn.models.dnn.DNN object at 0x7f979680f908>
>>> validate_tfl_audio_ann_model(am, BUZZ1_valid_X, BUZZ1_valid_Y)
0.45652173913043476
>>> validate_tfl_audio_ann_model(am, BUZZ2_valid_X, BUZZ2_valid_Y)
0.49333333333333335
>>> validate_tfl_audio_ann_model(am, BUZZ3_valid_X, BUZZ3_valid_Y)
0.2862029646522235
```

Training, Testing, and Validating ConvNet Audio Models with TFLearn

The file `tfl_audio_convnets.py` contains examples of how to construct and train, test, and validate ConvNets on the 3 audio datasets, persist them, and load them into Python. The interaction below shows you how to do these steps.

```
>>> from tfl_audio_convnets import *
>>> aud_cn = make_audio_convnet_model()
>>> train_tfl_audio_convnet_model(aud_cn, BUZZ1_train_X, BUZZ1_train_Y,
                                  BUZZ1_test_X, BUZZ1_test_Y)
---------------------------------
Run id: audio_cn_model
Log directory: /tmp/tflearn_logs/
---------------------------------
Training samples: 7000
Validation samples: 2110
--
Training Step: 700  | total loss: 0.56288 | time: 5.435s
| SGD | epoch: 001 | loss: 0.56288 - acc: 0.7489 | val_loss: 0.57145 -
                                                    val_acc: 0.7957 -- iter: 7000/7000
--
Training Step: 1400  | total loss: 0.62500 | time: 5.349s
| SGD | epoch: 002 | loss: 0.62500 - acc: 0.7327 | val_loss: 0.46812 -
                                                    val_acc: 0.8052 -- iter: 7000/7000
--
>>> validate_tfl_audio_convnet_model(aud_cn, BUZZ1_valid_X, BUZZ1_valid_Y)
0.4165217391304348

>>> aud_cn.save('/home/vladimir/teaching/AI/project_01/models/aud_cn.tfl')
>>> cnm = load_audio_convnet_model('/home/vladimir/teaching/AI/project_01/models/aud_cn.tfl')
>>> validate_tfl_audio_convnet_model(cnm, BUZZ1_valid_X, BUZZ1_valid_Y)
0.4156521739130435
```

Training, Testing, and Validating ConvNet Models with Keras

This part of the project will give you experience with Keras. The file `keras_image_convnets.py` contains examples of how to construct and train, test, and validate ConvNets on image datasets, persist them, and load them into Python. The interaction below shows you how to do these steps. We'll train them on one dataset `BEE3`, which is a 3-way classification – BEE, NO_BEE, SHADOW_BEE. The zip archive BEE3 contains the images required to train, test, and validate this model.

Here's how you can load all datasets from BEE3 with `keras_image_convents.py`.

```
>>> python3
Python 3.6.9 (default, Oct  8 2020, 12:12:24)
[GCC 8.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from keras_image_convnets import *

datasets/BEE3/training/bee/*/*.png
********** CHECK 0000 **********
loaded 0...
loaded 5000...
```

```
loaded 10000...
loaded 15000...
loaded 20000...
loaded 25000...
loaded 30000...
loaded 35000...
37811
37811
datasets/BEE3/testing/bee/*/*.png
********** CHECK 0000 **********
loaded 0...
loaded 5000...
loaded 10000...
datasets/BEE3/validation/bee/*/*.png
********** CHECK 0000 **********
loaded 0...
loaded 5000...
loaded 10000...
(37811, 64, 64, 3)
(37811, 3)
(12471, 64, 64, 3)
(12471, 3)
```

Then we can use `train_keras_model()` to create and train a ConvNet. The source code of this function is in `keras_image_convnets.py` which defines a shallow ConvNet and trains it for 5 epochs. This function trains and persists the model in `MODEL_PATH/keras_img_model.h5`, where MODEL_PATH is a path variable.

```
>>> train_keras_model()
...

Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 64, 64, 32)        896
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 32, 32, 32)        0
_________________________________________________________________
flatten (Flatten)            (None, 32768)             0
_________________________________________________________________
dropout (Dropout)            (None, 32768)             0
_________________________________________________________________
dense (Dense)                (None, 128)               4194432
_________________________________________________________________
dense_1 (Dense)              (None, 3)                 387
=================================================================
Total params: 4,195,715
Trainable params: 4,195,715
Non-trainable params: 0
_________________________________________________________________

Epoch 1/5
```

```
591/591 [==============================] - 32s 54ms/step -
loss: 2.6728 - accuracy: 0.6898 - val_loss: 0.6037 - val_accuracy: 0.8019
Epoch 2/5
591/591 [==============================] - 31s 53ms/step -
loss: 0.6009 - accuracy: 0.8112 - val_loss: 0.7506 - val_accuracy: 0.7300
Epoch 3/5
591/591 [==============================] - 32s 54ms/step -
loss: 0.5709 - accuracy: 0.8202 - val_loss: 0.5312 - val_accuracy: 0.8370
Epoch 4/5
591/591 [==============================] - 31s 53ms/step -
loss: 0.5235 - accuracy: 0.8384 - val_loss: 0.4797 - val_accuracy: 0.8533
Epoch 5/5
591/591 [==============================] - 32s 53ms/step -
loss: 0.5179 - accuracy: 0.8388 - val_loss: 0.7393 - val_accuracy: 0.7144
```

The function `load_keras_model()` loads the persisted model in Python. We can then use the function `evaluate_keras_model()` to evaluate this model on the BEE3 validation dataset.

```
>>> km = load_keras_model()
>>> km
<tensorflow.python.keras.engine.sequential.Sequential object at 0x7fe58c046a20>
>>> evaluate_keras_model(km)
Training Accuracy: 72.17%
Testing Accuracy: 71.44%
Validation Accuracy: 60.65%
```

Unit Tests

I've written three unit test files (`tfl_audio_uts.py`, `tfl_image_uts.py`, and `keras_image_uts.py`) that you can use to test your nets. Below are sample outputs on running my nets described above trained on the datasets for 2-5 epochs and persisted, which is why their validation accuracies are low.

Here's a sample output I got by runninng `python tfl_audio_uts.py`. I skip output lines with loading and TFLearn warning messages.

```
**** Ann valid. acc on BUZZ1 = 0.46695652173913044
**** Ann valid. acc on BUZZ2 = 0.416
**** Ann valid. acc on BUZZ3 = 0.3993728620296465
**** CN valid. acc on BUZZ1 = 0.38
**** CN valid. acc on BUZZ2 = 0.2803333333333333
**** CN valid. acc on BUZZ3 = 0.2767958950969213
```

Here's a sample output I got by runninng `python tfl_image_uts.py`. I skip output lines with loading and TFLearn warning messages.

```
**** Ann valid. acc on BEE1_1S_gray = 0.5751133786848073
**** Ann valid. acc on BEE2_1S_gray = 0.6627143378329077
**** Ann valid. acc on BEE4_gray = 0.5183423913043478
**** ConvNet valid. acc on BEE1_gray = 0.9302721088435374
**** ConvNet valid. acc on BEE2_1S = 0.6172017511856986
**** ConvNet valid. acc on BEE4 = 0.5905385375494071
```

Here's a sample output I got by runninng `python keras_image_uts.py` I skip output lines with loading and TFLearn warning messages.

```
Keras BEE4 valid accuracy: 83.80%
Keras BEE4 valid loss: 45.32%
```

What You Need to Do

1. Train one ANN network with TFLearn for `BEE1_gray`, `BEE2_1S_gray`, and `BEE_4_gray`. Persist your trained ANN in `image_ann.tfl`.

2. Train one ConvNet with TFLearn for `BEE1`, `BEE2_1S`, and `BEE_4`. Persist your trained CN in `image_cn.tfl`.

3. Train one ANN network with TFLearn for the 3 audio datasets: `BUZZ1`, `BUZZ2`, and `BUZZ3`. Persist your trained ANN in `aud_ann.tfl`.

4. Train one ConvNet for the 3 audio datasets with TFLearn: `BUZZ1`, `BUZZ2`, and `BUZZ3`. Persist your trained CN in `aud_cn.tfl`.

5. Train one ConvNet for the `BEE3` dataset with Keras and persist it into `keras_image_model.h5`.

What to Submit

These files are due on Canvas on 11/08 by 11:59pm on Canvas.

1. `image_ann.tfl` – your best image TFLearn ANN
2. Your source code for your best image TFLearn ANN in `tfl_image_anns.py`
3. `image_cn.tfl` – your best image TFLearn ConvNet;
4. Your source code for your best image TFLearn ConvNet in `tfl_image_convnets.py`
5. `aud_ann.tfl` – your best audio TFLearn ANN;
6. Your source code for your best audio TFLearn ANN in `tfl_audio_anns.py`
7. `aud_cn.tfl` – your best audio TFLearn CN;
8. Your source code for your best audio TFLearn CN in `tfl_audio_convnets.py`
9. `keras_image_model.h5` – your best image Keras ConvNet for BEE3;
10. Your source code for your best image Keras CN in `keras_image_convnets.py`
11. `CS5600_6600_F21_PRoject1.pdf` – your 3-5 page project report on what you've done; what worked and what didn't; lessons learned; accuracy graphs/tables; list of references (online materials, articles, books, papers, etc.) you used in your project.

How We'll Test Your Nets

Your nets will be tested on similar (but smaller) audio and image datasets with `tfl_audio_uts.py`, `tfl_image_uts.py`, and `keras_image_uts.py` You'll receive the performance report of your nets each dataset on which we'll test them.

Happy Hacking and Bulldozing!