Machine Learning Engineer Nanodegree

Capstone Project:

Detecting Emotions through Facial Expressions

My Nguyen

August 9th, 2017

I. Definition

Project Overview

Recent advancements in the field of computer vision, particularly image recognition, have given new waves of progress to the task of emotion detection through facial expressions. Convolutional Neural Networks (CNNs) have been in the center stage of these improvements, outperforming other methods. Notable examples include the Emotion in The Wild (EmotiW) challenges, where top submissions have consistently employed CNNs [1],[2],[3], and the Facial Expression Recognition (FER) competition from Kaggle, where the top three submissions all used CNNs [4]. The mentioned Kaggle competition is also the subject of this project.

This project is inspired by AutoTutor, an Intelligent Tutoring System (ITS) that is both cognitively and emotionally intelligent. It can carry conversations with students, know when the student is frustrated or bored and adjust its teaching methods accordingly [5]. It is my hope that CNNs can help the tutor better recognize emotions. Although learning emotions are not the same as the basic emotions represented in this Kaggle competition [5], due to the lack of labelled datasets of learning emotions, the FER Kaggle dataset is a good starting point.

Problem Statement

The task at hand is to build a deep learning model that looks at images of human faces and tells us the emotion that the person is feeling. Given recent successes of CNN at classifying images into categories, we will rely on one to tackle this problem. Our CNN model will take in inputs of raw pixel values and output a prediction of the

expressed emotion. There are seven emotions being considered for this dataset: angry, disgust, happy, sad, fear, surprise, and neutral. Before feeding the pixel values to the network, we will employ various data preprocessing and augmentation techniques. The network itself will have several convolutional layers, pooling layers, and fully connected layers. Regularization and drop out techniques will be explored as well as multiple choices of activation functions and classifiers, among others that will be discussed later on. Needless to say, this is a multi-class classification problem.

Metrics

As we will be comparing our results directly with those of Kaggle's competitors, we will use the same evaluation metrics as the one used in this competition: accuracy rate. It can be defined as $\frac{\#\text{ of correctly classified images}}{total\ number\ of\ images}$.

For completeness, it's worth pointing out that there are other metrics that can be highly relevant for this problem such as precision, recall, F1 score, or kappa because we're dealing with an imbalanced dataset. Depending on the end goal of the result, one can choose a metric that best suits their purposes. For example, if one is more concerned with the ability to correctly classify emotions than being able to pick out happy people among the crowd (to eliminate them because you're an evil robot who despises happiness), precision should be weighted more.

II. Analysis

Data Exploration

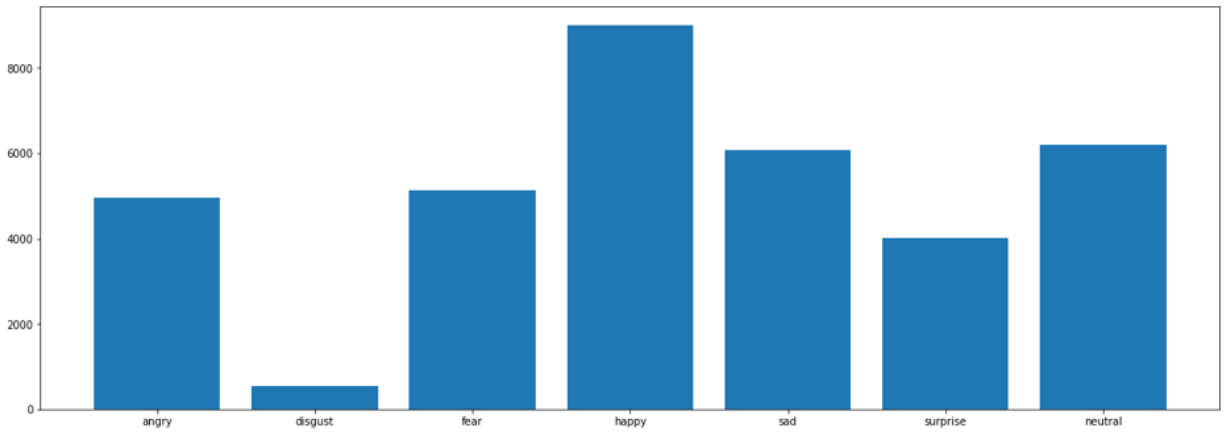
The dataset was highly curated by human labelers from a pool of google images. Faces were cropped out. Images were resized to 48x48 pixels, and converted to grayscale. There are 35887 images in the final dataset, with 4953 of "Anger", 547 of "Disgust", 5121 of "Fear", 8989 of "Happiness", 6077 of "Sadness", 4002 of "Surprise", and 6198 "Neutral". Further details about the dataset can be found here [6].

This dataset has the advantage of being spontaneous and realistic as opposed to posed and perfect lighting conditions as in laboratory settings. One disadvantage could be that the perceived emotions are not the same as expressed emotions.

Exploratory Visualization

Above is an example of images for each emotion. From left to right: angry, disgust, fear, happy, sad, surprise, and neutral [7]. As can be seen in the picture, there are blank images in the dataset. Therefore, one of the formatting steps is to remove blank images.

The dataset is highly imbalanced with happiness being the most depicted emotions, followed by neutral and sadness with approximately the same frequency. Not too far behind come fear and anger. Surprise is next, and disgust is last with less than 2% of total images. The histogram below shows the frequency distribution for each class.



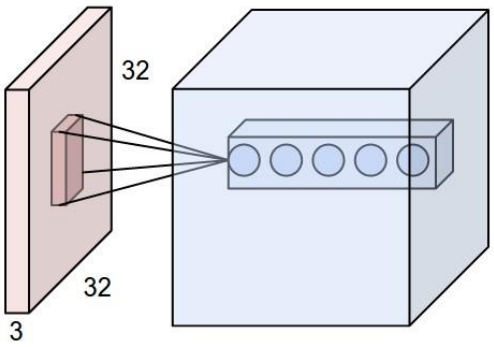
Imbalanced classes can lead to bad results in NN [8]. Depending on the problem, there have been proposed methods to combat this issue [9]. This project employed two techniques: oversampling the minority class and adjusting the weights of each class in the loss function, giving a higher loss to the under-represented classes. Unfortunately, none of them helped.

Algorithms and Techniques

A simple CNN consisted of three blocks of convolutional layer and pooling layer, one dropout layer, and one fully connected layer was implemented. Various data augmentation, weight initialization techniques, activation functions, classifiers, optimizers, and regularizers were explored.

Convolutional layers: Unlike regular neural networks, CNN takes advantage of the fact that the input to the network is an image, which has the shape of width x height x depth. The depth dimension usually refers to the color channel: one for grayscale images and three (red, blue, and green) for color images. Instead of mapping each input feature (or pixel, in the case of images) to a neuron in the hidden layer, CNN maps a small area (width x height x depth) of the image to one "pixel" (or neuron) of the hidden layer through what's called a filter, or kernel. Think of each filter as a "square magnifying glass" that moves methodically through the input: left to right, top to bottom one "inch" at a time (or one pixel at a time, or how many pixels at a time depending on the "stride" parameter). The size of the glass is determined by the size of the filter being implemented, and the fact that it's the same glass going through the whole image makes it that the hidden neurons share the same parameters. We can use as many glasses that we'd like, which makes up the depth of the convolutional layer.

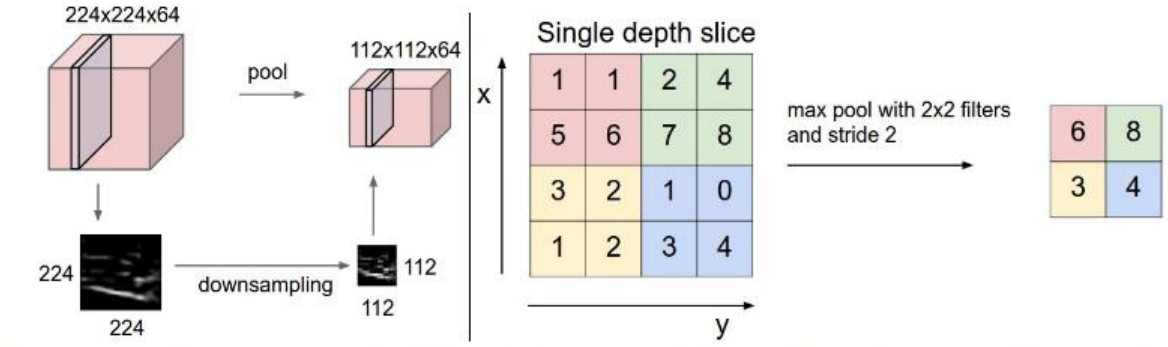
See [10] for a visual of "convolving a magnifying glass/filter through an image":



Source: http://cs231n.github.io/convo 1

To the left is an example input volume in red (e.g. a 32x32x3 CIFAR-10 image), and an example volume of neurons in the first Convolutional layer. Each neuron in the convolutional layer is connected only to a local region in the input volume spatially, but to the full depth (i.e. all color channels). There are multiple neurons (5 in this example, or 5 glasses in the "magnifying glass" terminology) along the depth, all looking at the same region in the input.

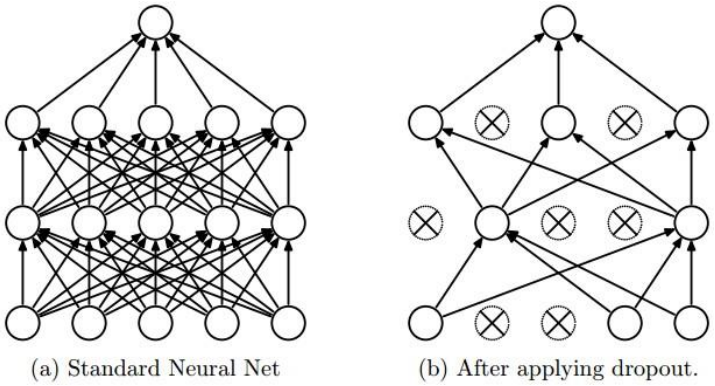
Pooling layers act as a "shrinking glass" that convolves over its inputs and reduce their size, usually by half. This is done to overcome overfitting. Pooling layers are commonly placed after convolutional layers.



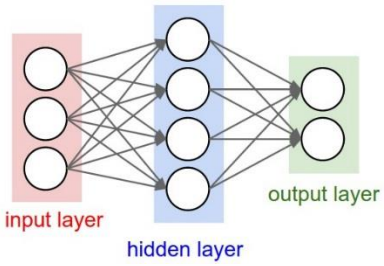
Pooling layer downsamples the volume spatially, independently in each depth slice of the input volume. **Left:** In this example, the input volume of size [224x224x64] is pooled with filter size 2, stride 2 into output volume of size [112x112x64]. Notice that the volume depth is preserved. **Right:** The most common downsampling operation is max, giving rise to **max pooling**, here shown with a stride of 2. That is, each max is taken over 4 numbers (little 2x2 square).

Source: http://cs231n.github.io/convolut 1

Dropout layers: This is a simple but another effective way to combat overfitting, in which neurons and their associated connections are dropped randomly based on a probability parameter. This sounds counter-intuitive but it has the effect of forcing the network to learn features that are robust and abundant enough that even if some neurons were to be missing, the rest of the network can still carry the information.



Source: http://cs231n.github.io/neural-n 1



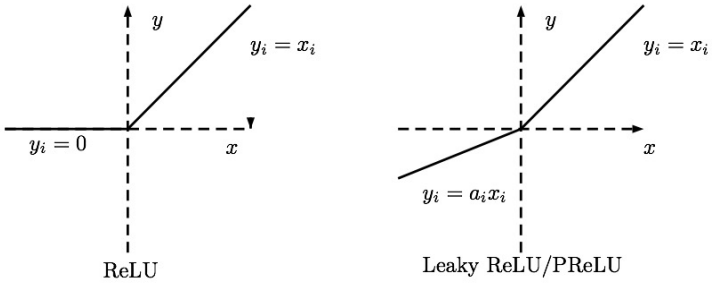
http://cs231n.github.io/neural-networks- 1

Fully connected layers: These are hidden layers from a regular neural network, in which each neuron in the hidden layer connects to each feature in its input space.

Weight initializations: Even though a small detail, wrong initialization of a network can lead to slow or no learning at all. Here, small random numbers from a truncated normal distribution where large deviations from the mean are truncated were used.

Kernel regularization is another way to counter overfitting where large values of weight parameters are penalized through the loss functions. Thus, small weights are preferred so that no one features imposes an over-powering effect over other features.

Activation functions are the "wiggles" of neural networks where non-linearity exists. Without the use of non-linear activation functions, the whole network can be squashed down to one single dot product of the weight and input vectors, thus eliminating the point of using a complex model. This project tried both ReLU and LeakyReLU functions with little difference in the result.



Classifiers do the final job of a neural network. They put things into categories. The most intuitive classifier that is commonly being used is softmax. It transforms real numbers into a range of 0-1 which can be interpreted as probabilities. The class with the highest probability is the final predicted class of the input in question. Another commonly used classifier is Support Vector Machine where, through its loss function, it tries to make sure classes are linearly separated by a safe margin.

Optimizers are the computational engines that make all this possible. The most commonly used optimizer is stochastic gradient descent, where at each step, through backpropagation, weights and biases are updated in the direction that would lower the loss. These directions are determined by the gradients of the loss function at its current state. Another optimizer that was used in this project is Adam. For details about Adam, please see [11].

Benchmark

I hope to be among the top 10 of the leaderboard, ie. minimum of 62% accuracy.

III. Methodology

Data Preprocessing

The following preprocessing steps were performed:

- Read in data from csv file.
- Remove blank images.
- Convert data into numpy arrays and reshape the pixels to (n_example, 48, 48, 1), and labels to (n_example, 1).

- Split data into train, validation, and test sets. As the whole dataset was released, I used the public leaderboard set as the validation set, and private leaderboard as the test set.
- One hot encode label arrays (y_train, y_validation, & y_test).
- Numerous combinations of data augmentation techniques were tried: feature wise center & normalization, sample wise center & normalization, horizontal flip, rotation, height shift, width shift, rescaling, etc. The final chosen methods were feature wise center & normalization, and horizontal flip as they gave the best results on the validation set.

Implementation

Using the 1st place solution as a starting point [7], a 62% accuracy rate was obtained by following the CNN structure, and two simple augmentation techniques: feature wise center and standardization, and mirror images.

Below is the original network structure that was implemented in Keras 2.0.6 on Tensorflow backend for this project:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
Block1_Crop (Cropping2D)     (None, 42, 42, 1)         0
_________________________________________________________________
Block1_Conv (Conv2D)         (None, 42, 42, 32)        832
_________________________________________________________________
Block1_Pool (MaxPooling2D)   (None, 21, 21, 32)        0
_________________________________________________________________
Block2_Pad (ZeroPadding2D)   (None, 23, 23, 32)        0
_________________________________________________________________
Block2_Conv (Conv2D)         (None, 20, 20, 32)        16416
_________________________________________________________________
Block2_Pool (AveragePooling2 (None, 10, 10, 32)        0
_________________________________________________________________
Block3_Conv (Conv2D)         (None, 10, 10, 64)        51264
_________________________________________________________________
Block3_Pool (AveragePooling2 (None, 5, 5, 64)          0
_________________________________________________________________
Block4_Flat (Flatten)        (None, 1600)              0
_________________________________________________________________
Block4_Dropout (Dropout)     (None, 1600)              0
_________________________________________________________________
Block4_Dense (Dense)         (None, 3072)              4918272
_________________________________________________________________
Block4_Out (Dense)           (None, 7)                 21511
=================================================================
Total params: 5,008,295
Trainable params: 5,008,295
Non-trainable params: 0
```

Stochastic gradient descend was implemented with Nesterov momentum of 0.9, and a learning rate decay logic that halves the learning rate when validation loss hasn't decreased for five epochs. Categorical cross-entropy loss was used. Batch size was 256. The model ran for about 100 epochs and was stopped when the validation loss has no longer improved for 41 epochs, which gives the learning rate a chance to be lowered four times.

One challenge I ran into when first started was that my model wasn't learning: loss wasn't decreasing, accuracy rate wasn't increasing. I later found out that rescaling the pixel values by diving them by 255 was the root of the problem. My guess is that by diving by 255, the input values became so small that none of the weight updates managed to move the model anywhere. This is possibly a case of the vanishing gradients problem.

Another problem I came across was implementing tensorboard. It turns out that tensorboard does not generate histograms or distributions when a generator is used for generating validation data [12]. In my case it was a simple fix as I only needed to center and normalize my data so I did it manually (not through Keras) before feeding the data to model.fit method.

Python code can be found in the main script fer.ipynb and helper.py.

Refinement

To get better results. I tried numerous variants of data augmentation, optimizer engines, weight initializers, batch normalization, activation functions, slight modifications to the network structure, output layer classifiers, and others. In the end, the only two additions that provided significant and consistent improvements are L2 regularization with alpha of 0.001, which provided an improved accuracy of 1%-2% on the validation set, and an average of 5 models for testing, which provided an improvement of ~2%.

For a full list of procedures that were tried, please see the Appendix.

IV. Results

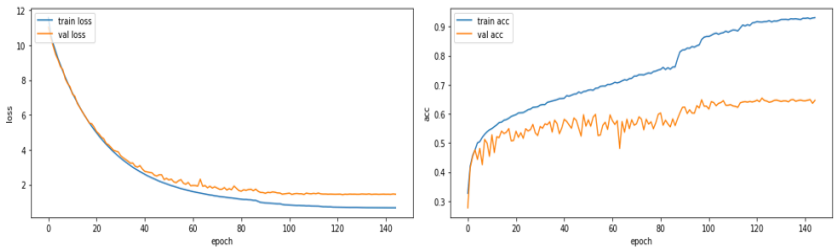
Data augmentation chosen was feature wise center and standardization, and mirror images. L2 regularization was applied with alpha of 0.001. Five models were run and their predictions were averaged to provide final outputs. Each of the five models obtained an accuracy rate of 64% - 65% on the test set (private leaderboard) while their average achieved 67.17%, which would have placed 5th on the competition.

The final model has the following structure:

Layer (type)	Output Shape	Param #
block1_crop (Cropping2D)	(None, 42, 42, 1)	0
block1_conv (Conv2D)	(None, 42, 42, 32)	832
block1_bn (BatchNormalizatio	(None, 42, 42, 32)	128
block1_relu (LeakyReLU)	(None, 42, 42, 32)	0
block1_pool (MaxPooling2D)	(None, 21, 21, 32)	0
block2_pad (ZeroPadding2D)	(None, 23, 23, 32)	0
block2_conv (Conv2D)	(None, 20, 20, 32)	16416
block2_bn (BatchNormalizatio	(None, 20, 20, 32)	128
block2_relu (LeakyReLU)	(None, 20, 20, 32)	0
block2_pool (AveragePooling2	(None, 10, 10, 32)	0
block3_conv (Conv2D)	(None, 10, 10, 64)	51264
block3_bn (BatchNormalizatio	(None, 10, 10, 64)	256
block3_relu (LeakyReLU)	(None, 10, 10, 64)	0
block3_pool (AveragePooling2	(None, 5, 5, 64)	0
block4_flat (Flatten)	(None, 1600)	0
block4_dropout (Dropout)	(None, 1600)	0
block4_fc (Dense)	(None, 3072)	4918272
block4_relu (LeakyReLU)	(None, 3072)	0
block4_out (Dense)	(None, 7)	21511
block4_softmax (Activation)	(None, 7)	0

Total params: 5,008,807
Trainable params: 5,008,551
Non-trainable params: 256

Below is the loss and accuracy during training and validation:

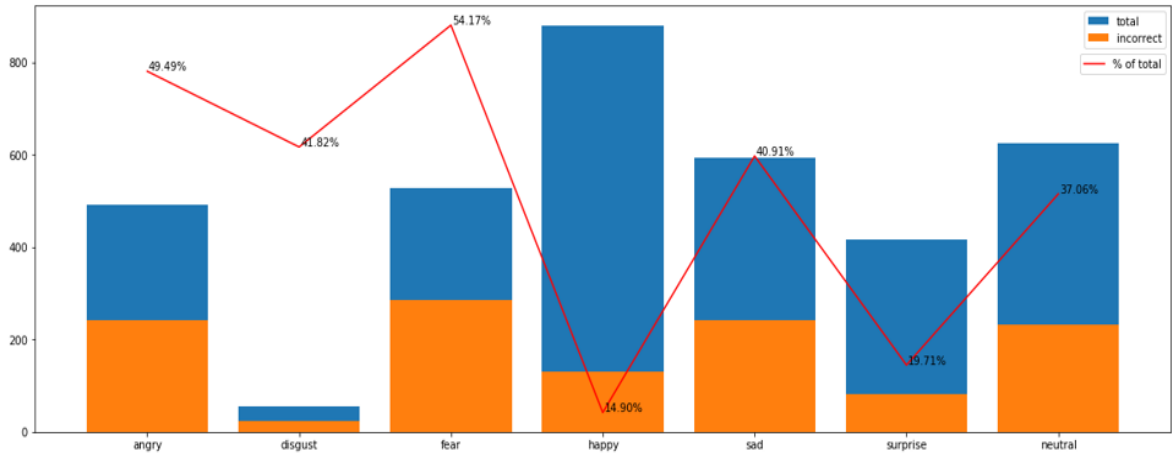


Overfitting seemed to still be present. Increasing alpha of L2 regularization helped with this problem but decreased the accuracy rate. Changing the dropout rate did not have much of an effect to the model.

V. Conclusion

Free-Form Visualization

A histogram of the emotions that were incorrectly predicted vs total shows that the model is better at classifying certain emotions than others. Specifically, the model can predict ‘happy’ very well but not ‘fear’ or ‘angry’. This could be due to the fact that there are more ‘happy’ images to train with. However, artificially increasing the sample size of other classes by augmentation or oversampling the underrepresentative classes did not help. One reason can be that the mentioned techniques did not increase the variety of the data thus did not increase the generalization ability of the model.



Below is an example of images that the model predicts correctly and incorrectly along with its softmax output.

Reflection

Through this project, a CNN was implemented to classify human emotions by their facial expressions. Many implementation details were tried out and their effects were noted on this particular dataset. It was a great learning experience for me in which methods that were thought to be beneficial turned out to not making a difference (such as more data augmentation: rotation, shift; balance imbalanced classes), whereas details that seem unimportant (rescaling inputs) were actually crucial. I took it to heart that the architecture of a network is one thing, training it is another. Even with a "best" network, bad implementation can lead to very bad results, or no learning at all. As the saying goes, "the devil is in the details".

Improvement

It would be very interesting to see if (1) a random optimized search for the combinations of the ensemble rather than a simple average, (2) a deeper network, (3) transfer learning from established nets (such as VGGs, ResNet, etc.), or (4) fractional max pooling [13] (which is the most state-of-the-art on CIFAR-10 [14]) could produce better results.

Appendix

Various combinations of the following techniques were tried on the dataset:

- Data augmentation
- Batch normalization
- Balancing class by:
    - Using class_weight argument from Keras
    - Over sampling minority class, method from package imblearn
- Optimizers:
    - Stochastic gradient descend with momentum and learning rate decay
    - Adam
- Activation: ReLU vs LeakyReLU
- Keras callback settings of ReduceLROnPlateau, ModelCheckpoint, and EarlyStopping
- Slight modifications to the structure of the network:
    - Add one convolution layer of 3x3 filter size at the beginning
    - Add one dense layer of 1000 neurons before output layer
    - Different drop out rate, both higher and lower than 20%
- Classifiers:
    - Softmax vs SVM
- Ensembles

References:

[1] Z. Yu and C. Zhang, "Image based static facial expression recognition with multiple deep network learning," in Proceedings of the 2015 ACM on International Conference on Multimodal Interaction, ICMI '15, (New York, NY, USA), pp. 435–442, ACM, 2015.
[2] B. Kim, J. Roh, S. Dong, and S. Lee, "Hierarchical committee of deep convolutional neural networks for robust facial expression recognition," Journal on Multimodal User Interfaces, pp. 1–17, 2016.
[3] Y. Fan, X. Lu, D. Li, and Y. Liu, "Video-based Emotion Recognition Using CNN-RNN and C3D Hybrid Networks," Proceedings of the 18th ACM International Conference on Multimodal Interaction, pp. 445-450, ACM, 2016.
[4] https://www.kaggle.com/c/challenges-in-representation-learning-facial-expression-recognition-challenge
[5] D'Mello, S. K., Graesser, A. C.. 2012. AutoTutor and Affective AutoTutor: Learning by Talking with a Cognitively and Emotionally Intelligent Computer that Talks Back. ACM Trans. Interactive Intelligent. Syst. X, X, Article XX (Month 2011), X pages.
[6] [1307.0414] Challenges in Representation Learning: A report on three machine learning contests https://arxiv.org/abs/1307.0414
[7] Yichuan Tang, 2015. "Deep Learning using Linear Support Vector Machine".
[8] The Impact of Imbalanced Training Data for Convolutional Neural Networks
[9]
http://machinelearningmastery.com/tactics-to-combat-imbalanced-classes-in-your-machine-learning-dataset/
[10] https://udacity-reviews-uploads.s3.amazonaws.com/_attachments/19273/1502304238/Convolution_schematic.gif
[11] https://arxiv.org/abs/1412.6980

[12] https://github.com/fchollet/keras/issues/3358
[13] Benjamin Graham, 2015. "Fractional Max-Pooling".
[14]
http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html#43494641522d313030